# Intelligent Mobile Robot Learning in Autonomous Navigation

Chen Xia

*N° d'ordre*: | 2 | 8 | 0 |

**ÉCOLE CENTRALE DE LILLE**

# THÈSE

présentée en vue d'obtenir le grade de

# DOCTEUR

en

*Spécialité : Automatique, Génie Informatique, Traitement du Signal et des Images*

par


## Chen XIA

Master of Science of Beihang University (BUAA)


**Doctorat délivré par l'École Centrale de Lille**

Titre de la thèse :

## Apprentissage Intelligent des Robots Mobiles dans la Navigation Autonome

Soutenue le 24 novembre 2015 devant le jury d'examen :

| | | |
|---|---|---|
| M. Pierre BORNE | École Centrale de Lille | Président |
| M. Noureddine ELLOUZE | École Nationale d'Ingénieurs de Tunis | Rapporteur |
| M. Dumitru POPESCU | Université Polytechnique de Bucarest | Rapporteur |
| M. Abdelkader EL KAMEL | École Centrale de Lille | Directeur de thèse |
| M. Khaled MELLOULI | IHEC Carthage, Tunisie | Examinateur |
| Mme. Shaoping WANG | Beihang University, Chine | Examinateur |
| Mme. Liming ZHANG | University of Macau | Examinateur |


Thèse préparée dans le Centre de Recherche en Informatique, Signal et Automatique de Lille (CRIStAL), UMR CNRS 9189 - École Centrale de Lille
École Doctorale SPI 072
**PRES Université Lille Nord-de-France**

*Serial N°* : | 2 | 8 | 0 |

**ÉCOLE CENTRALE DE LILLE**

## THESIS

Presented to obtain the degree of

## Doctor of Philosophy

in

Topic : Automatic Control, Computer Science, Signal and Image Processing

by


# Chen XIA

Master of Engineering of Beihang University (BUAA)


**Ph.D. awarded by École Centrale de Lille**

Title of the thesis :

## Intelligent Mobile Robot Learning in Autonomous Navigation

Defended on November 24, 2015 in presence of the committee :

| | | |
|---|---|---|
| Mr. Pierre BORNE | École Centrale de Lille | President |
| Mr. Noureddine ELLOUZE | École Nationale d'Ingénieurs de Tunis | Reviewer |
| Mr. Dumitru POPESCU | Université Polytechnique de Bucarest | Reviewer |
| Mr. Abdelkader EL KAMEL | École Centrale de Lille | Supervisor |
| Mr. Khaled MELLOULI | IHEC Carthage, Tunisie | Examiner |
| Mrs. Shaoping WANG | Beihang University, Chine | Examiner |
| Mrs. Liming ZHANG | University of Macau | Examiner |


Thesis prepared within the Centre de Recherche en Informatique, Signal et Automatique de Lille (CRIStAL), UMR CNRS 9189 - École Centrale de Lille
École Doctorale SPI 072
**PRES Université Lille Nord-de-France**

*To my parents,*
*to all my family,*
*to my professors,*
*and to my friends.*

# Acknowledgements

This dissertation has been realized at “Centre de Recherche en Informatique, Signal et Automatique de Lille (CRIStAL)” in École Centrale de Lille, with the research group “Optimisation : Modèles et Applications (OPTIMA)”, from September 2012 to November 2015. This work is financially supported by China Scholarship Council (CSC).

First and foremost, I offer my sincerest gratitude to my supervisor, Prof. Abdelkader El Kamel. He has provided his supervision, valuable guidance, continuous encouragement as well as given me extraordinary experiences throughout my Ph.D. experience. A special acknowledgment should be shown to Prof. Shaoping Wang, who enlightened me at the first glance of research, and she always supported my thesis work during the past three years with her helpful suggestions and discussions. My thanks to both Prof. El Kamel and Prof. Wang for having involved me in their research cooperation project and giving me this opportunity to study in France.

Besides my supervisor, I would like to thank Prof. Pierre BORNE for his kind acceptance to be the president of my Ph.D. Committee, as well as Prof. Noureddine ELLOUZE and Prof. Dumitru POPESCU, who have kindly accepted the invitation to be reviewers of my Ph.D. thesis, for their encouragement, insightful comments and hard questions. My gratitude to Prof. Khaled MELLOULI, Prof. Shaoping Wang and Prof. Liming ZHANG, for being the examiners of my thesis and for their kind acceptance to take part in the jury of the Ph.D. defense.

I am also very grateful to the staff in École Centrale de Lille. Vanessa Fleury, Christine Yvoz, and Brigitte Foncez have helped me in the administrative work. Many thanks go also to Patrick Gallais, Gilles Marguerite, Jacques Lasue, for their kind help and hospitality. Special thanks go to Christine Vion, Martine

Mouvaux for their support in my lodgment life at the dormitory "Léonard de Vinci".

My sincere thanks also goes to Dr. Tian Zheng, Dr. Yue Yu and Dr. Daji Tian, for offering me the useful advices during my study in the laboratory as well as after their graduation.

I would like to take the opportunity to express my gratitude and to thank my fellow workmates in CRIStAL: Bing Liu, Yihan Liu, Qi Sun, Jian Zhang for the stimulating discussions for the hard teamwork. Also I wish to thank my friends and colleagues: Hongchang Zhang, Yu Du, Lijie Bai, Qi Guo, Jing Bai, Lijuan Zhang, Youwei Dong, Ben Li, Xiaokun Ding, etc., for all the fun we have had in the past three years. All of them have given me support and encouragement in my thesis work.

All my gratitude goes to Ms. Hélène Catsiapis who taught me the French language and culture. My knowledge and interest in the culture is inspired by her work and enthusiasm. She organized many interesting and unforgettable trips in France, and she managed to open my appetite for art, history, gastronomy and wine.

My Ph.D. program was supported by the cooperation between China and Scholarship Council (CSC) and Ecoles-Centrales Intergroup. So I would like to thank all of the people who have built this relationship and contributed to select candidates for interesting research projects.

Last but not least, I convey special acknowledgment to my family for supporting me to pursue this degree and to accept my absence for four years of living abroad: my parents Baoqing Xia and Guihong Gong, for giving birth to me at the first place and supporting me spiritually throughout my life. So as to the rest of my family for their love and support. Their encouragement and understanding during the course of my dissertation made me pursue the advanced academic degree.

Villeneuve d'Ascq, France *Chen Xia*
November, 2015

# Abstract

Modern robots are designed for assisting or replacing our human beings to perform complicated planning and control operations and tasks, such as manipulating objects, assisting experts in a variety of professions, navigating in outdoor environments, exploring unknown territories, and driving in urban areas. Designing a control schema for such robots to execute these tasks is usually a complicated process, even for people specialized in programming robots, which requires creating by hand a new and different controller for each particular task. The designer has to take deliberately the wide range of situations that the robot may face into account. This sort of manually programming is generally an expensive as well as intense time-consuming process. Rather than pre-programming a robot for all the tasks, it would be more useful if the robot could learn such tasks by themselves.

This dissertation focuses on the intelligent robot control in autonomous navigation tasks and investigates the robot learning in three aspects.

First, we consider the robot learning from expert demonstrations. This method is inspired by the human instinct of imitation. Providing the examples of standard behaviors to the robots, the robots learn from these data and generalize over all potential situations that are not given in the examples. We embedded an inference mechanism by applying neural network into the robot controllers. With an acceptable number of demonstrations, the robot can acquire the independent navigation skills.

Second, we consider the robot self-learning ability without expert demonstrations in autonomous navigation. We use the state-of-the-art reinforcement learning techniques to train the robot via interaction with the robots. A neural network is also incorporated to play the role of fast generalization. We train the robot with all the past state-action pairs collected during the process of interaction, and this can help the learning to converge in a number of episodes that is

greatly smaller than the traditional methods.

Third, we consider the robot learning the potential rewards in states via inverse reinforcement learning. Given the expert demonstration, the robots should not only learn to pair the states and actions, but also try to understand the underlying framework of the demonstrations, the rewards. We proposed a nonlinear neural policy representation, and the max-margin inverse reinforcement learning algorithm is applied to refine and train the policy. The resulting policy can be used for robots to undertake autonomous navigation tasks.

Experimental results on these three aspects showed a reliable and robustness performance of robot autonomous navigation tasks by using our proposed methods. Therefore, we prove that the trend of robot learning instead of traditional robot programming will have a bright future and bring us more benefits and serve us better.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Abbreviations

$\mathbb{N}$ - Natural numbers
$\mathbb{R}$ - Real numbers
AGV - Automated guided vehicle
AMR - Autonomous mobile robot
ANN - Artificial neural network
AUV - Autonomous underwater vehicle
BPNN - Backpropagation neural network
CNN - Convolutional neural network
DP - Dynamic programming
FFNN - Feedforward neural network
GPI - Generalized policy iteration
IRL - Inverse reinforcement learning
LfD - Learning from demonstration
MAP - Maximum a posteriori
MC - Monte Carlo
MDP - Markov decision process
ML - Maximum likelihood
NN - Neural network
NNQL - Neural network based Q-learning
PI - Policy iteration
POMDP - Partial observable Markov decision process
RL - Reinforcement learning
SGD - Stochastic gradient descent
TD - Temperal difference
UAV - Unmanned aerial Vehicle
UGV - Unmanned ground vehicle

VI - Value iteration

# Chapter 1

# Introduction

## Contents



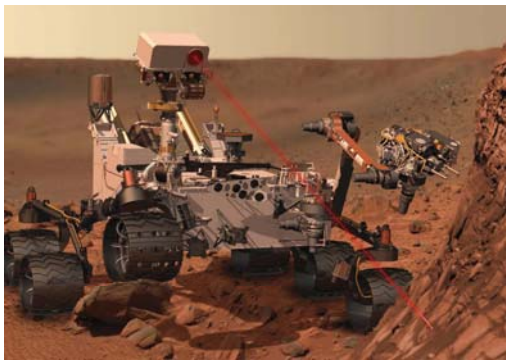
## 1.1 Background and Motivation

### 1.1.1 Mobile Robotics

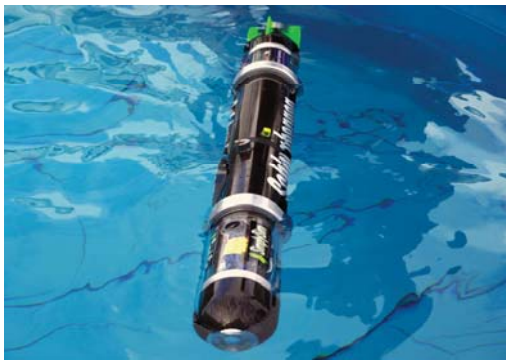
Robots are rapidly developing from industrial environments, which are physically fixed to their working places, to increasingly complex machines capable of performing challenging tasks in our daily environment. Traditional industrial robots used in manufacturing plants* where the environment is highly controlled are usually more-or-less stationary.

By contrast, mobile robots are the robotic systems that can operate in unconstrained environments and have the capability to move around freely using,

*This kind of robots are often referred to as robotic arms or manipulators.



(a) *Curiosity*: NASA's Mars exploration Rover.



(b) *Blackghost*: an AUV designed to undertake an underwater assault course autonomously with no outside control.

(c) *Kiva* Robots: that Amazon used to help employees in their warehouses.

(d) *RP-VITA*: a medical robot that doctors can make remote visits to patients via PCs or iPads running the robots.

Figure 1.1: Various applications of mobile robots.

for example, wheels. They can operate autonomously in a partially unknown and unpredictable environment without the need for physical or electro-mechanical guidance devices (autonomous mobile robot (AMR)). Alternatively, mobile robots can rely on guidance devices that allow them to travel a predefined navigation route in relatively controlled space (automated guided vehicle (AGV)).

Mobile robots have been widely used in various fields, such as space exploration (see Figure 1.1(a)), under water survey (see Figure 1.1(b)), industrial and military industries (see Figure 1.1(c)), and medical service applications (see Figure 1.1(d)), and so on.

In this dissertation, we place our focus on autonomous mobile robots, robots that are capable of making their own decisions depending on the situation at

hand rather than merely executing a pre-defined sequence of motions. In fact, since most robots equipped with such decision-making capabilities are mobile, we may treat an autonomous mobile robot as a mobile robot with the ability to make decisions.

Advances in autonomous mobile robots also have provided solutions to complex tasks previously only considered achievable by humans. Such domains include planetary or underwater exploration (Helmick *et al.*, 2006; Kunz *et al.*, 2008), operation in urban environments (Bohren *et al.*, 2008) and unmanned flight (Fabiani *et al.*, 2007).

For a comprehensive understanding of autonomous mobile robots, readers are recommended to see Siegwart *et al.* (2011); Tzafestas (2013) as references.

### 1.1.2 Autonomous Navigation

A fully autonomous robot can gain information about the environment, work for an extended period without human intervention, move either all or part of itself throughout its operating environment without human assistance, avoid situations that are harmful to people, property. An autonomous robot may also learn or gain new knowledge like adjusting for new methods of accomplishing its tasks or adapting to changing surroundings. Therefore, mobile robots need to have the capabilities of autonomy and intelligence, and to design algorithms that allow the robots to function autonomously in unstructured, dynamic, partially observable, and uncertain environments pose a challenge to researchers to deal with key issues such as uncertainty (in both sensing and action), reliability, and real-time response.

In each of those domains of mobile robot applications, mobility is almost pointless without the ability to navigate. Random movement, which does not require a navigation capability, may be useful for certain surveillance or cleaning operations, but for most scientific or industrial applications of mobile robots the ability to move in a purposeful manner is required. Hence, autonomous navigation plays a key role in the success of the robots, and also the baseline for the relative technologies of autonomous mobile robots.

A mobile robot navigation task refers to plan a path with obstacle avoidance to a specified goal and to execute this plan based on sensor readings. Mobile robots navigation roughly includes the following six interrelated competences:

1. Perception: to obtain and interpret sensory information;
2. Exploration: the strategy that guides the robot to select the next direction to go;
3. Mapping: to construct a spatial representation or an environment model by using the sensory information perceived;
4. Localization: the strategy to estimate the robot position within the spatial map that occurs simultaneously to navigation control;
5. Path planning: the strategy to find a path towards a goal location being optimal or not;
6. Path execution: to determine and adapt motor actions to environmental changes, also including obstacle avoidance.

A robot requires a mechanism that allows it to move freely in the environment, i.e., it must be able to detect and react to situations. It is the robot sensors that play such a role as the eyes of robots, and the robot know where it is or how it get to some place, or to be able to reason about where it has gone. The sensors can be flexible and mobile to measure the distance that wheels have traveled along the ground, to measure inertial changes and external structure in the environment. Knudson & Tumer (2011) summarized that the sensors may be roughly divided into two classes: internal state sensors, such as accelerometers, gyroscope, which provide the internal information about the robot's movements, and external state sensors, such as laser, infrared sensors, sonar, and visual sensors, which provide the external information about the environment. The data from internal state sensors may provide position estimates of the robot in a 2D space. The data from external state sensors may be used to directly recognize a place or a situation, or be converted to information in a map of the environment. In most cases, the sensor readings are imprecise and unreliable due to the noises. Therefore, it is important for the mobile robot navigation to process the sensor data with noises. Since neural networks have many processing nodes, each with primarily local connections, they may provide some degree of robustness or fault tolerance for interpretation of the sensor data.

To sum up, the autonomous navigation task is the ability for mobile robots to obtain enough information about the environment, process it, and act, moving safely through this environment, usually according to a predefined path. The ability to sense the surrounding environment is a fundamental requirement to any autonomous system. All action decisions are made based on what sensory inputs the robot perceive.

### 1.1.3 Robot Learning

Autonomous robots cannot always be programmed to execute predefined actions because one does not always know in advance the unpredicted situations that the robot might encounter. Today, however, most robots used in the industry are preprogrammed and require a well-defined and controlled environment. Reprogramming such robots is often a costly process requiring an expert. By enabling robots to learn tasks either through autonomous self-exploration or through guidance from a human teacher, robot installation and task reprogramming are simplified. Meanwhile, robots that cannot learn lack one of the most interesting aspects of intelligence. Recent researches has shown a drift toward artificial intelligence approaches to improve the robot autonomous ability based on accumulated experiences, and artificial intelligence methods can be computationally less expensive than classical ones. Machine learning approaches are often applied, to each the burden on system engineers. Learning therefore has become a central topic in modern robotics research.

Robot learning is a research field at the intersection of machine learning and robotics. It studies techniques allowing a robot to acquire novel skills or adapt to its environment through learning algorithms.* The embodiment of the robot, situated in a physical embedding, provides at the same time specific difficulties (e.g. high-dimensionality, real time constraints for collecting data and learning) and opportunities for guiding the learning process.

Robot learning consists of a multitude of machine learning approaches, particularly reinforcement learning, imitation learning, inverse reinforcement learning, and regression methods, that have been adapted sufficiently to domain so that they allow learning in complex robot systems such as helicopters, flapping-wing

*https://en.wikipedia.org/wiki/robot_learning

flight, legged robots, anthropomorphic arms and humanoid robots. While classical artificial intelligence-based robotics approaches have often attempted to manually generate a set of rules and models that allows the robot systems to sense and act in the real-world, robot learning centers around the idea that it is unlikely that we can foresee all interesting real-world situations sufficiently accurate.

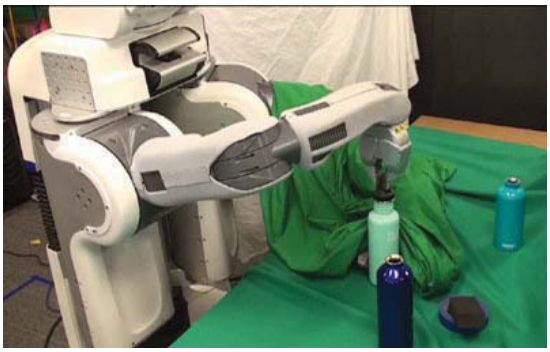
While robot learning covers a wide range of fields, from learning to perceive, to plan, to make decisions, etc., we focus our work on learning control in simulated or actual physical robots. In general, learning control refers to the process of acquiring a particular control system and a particular task by trial and error (Schaal & Atkeson, 2010). Reinforcement learning (RL) and learning from demonstration (LfD) are mentioned as two popular families of algorithms for learning policies for sequential decision problems (Cobo *et al.*, 2014).

Reinforcement learning algorithms solve sequential decision problems posed as Markov decision processes (MDPs), learning a policy by letting the agent explore the effects of different actions in different situations while trying to maximize a sparse reward signal. RL has been successfully applied to a variety of scenarios.

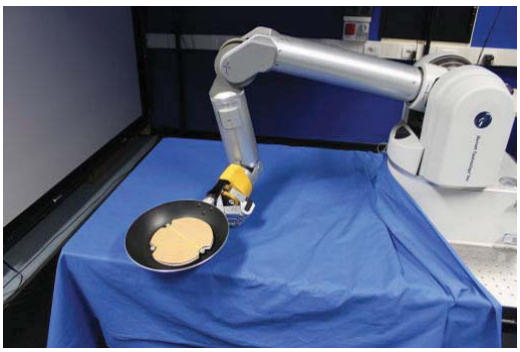
Learning from demonstration is an approach to robot/agent learning that takes as input demonstrations from a human in order to build action or task models. There are a broad range of approaches that fall under the umbrella of LfD research(Argall *et al.*, 2009). These demonstrations are typically represented as state-action tuples, and the LfD algorithm learns a policy mapping from states (input) to actions (output) based on the examples seen in the demonstrations. Inverse reinforcement learning (IRL), as one important branch of LfD methods, addresses the problem of estimating the reward function of an agent acting in a dynamic environment.

Another approach is to provide a mapping from sensory inputs to actions that statistically capture the key behavioral objectives without needing a model or detailed domain knowledge (Cummins & Newman, 2007). Such methods are well-suited to domains where the tools available to learn from past experience and adapt to emergent conditions are limited.

With the advent of increasingly efficient robot learning methods, one can observe a growing number of successful applications in robotics, such as autonomous helicopter control (Abbeel *et al.*, 2007, 2010; Ng *et al.*, 2006), self-driving car (Montemerlo *et al.*, 2008; Thrun *et al.*, 2006; Urmson *et al.*, 2008), autonomous



(a) BRETT is learning how to screw a cap onto a water bottle.



(b) Learning to flip an artificial pancake.

Figure 1.2: Various applications of robot learning.

underwater vehicles (AUVs) control (Carreras *et al.*, 2005), mobile robot navigation (Jaradat *et al.*, 2011), robot soccer control (Riedmiller *et al.*, 2009).

Recently, several interesting applications have appeared. Levine *et al.* (2015) worked with a Willow Garage Personal Robot 2 (PR2), named Berkeley Robot for the Elimination of Tedious Tasks (BRETT), and empowered BRETT has acquired the ability to learn to perform various tasks on its own via trial and error, without pre-programmed details about its surroundings. Those tasks include assembling a wheel part onto a toy airplane, stacking a Lego block, and screwing a cap on a water bottle (see Figure 1.2(a)). Mülling *et al.* (2013) used imitation and reinforcement learning techniques to enable a Barrett WAM arm to learn successful hitting movements in table tennis. Kormushev *et al.* (2010) taught a robot to flip a pancake. (see Figure 1.2(b)). Other successful robot learning applications also include Kolter & Ng (2009); Ratliff *et al.* (2009b).

### 1.1.4 Motivation

The ability of autonomous navigation plays an important role in the state-of-the-art self-driving cars. Dating back to 2003, the Defense Advanced Research Projects Agency (DARPA) of US government launched the "Grand Challenge" to spur the development of technologies needed to create the first fully autonomous ground vehicles capable of completing a substantial off-road course within a limited time. The Challenge required robotic vehicles to navigate a 142-mile long course through the Mojave desert in no more than 10 h. The first competition

was held on March 13, 2004. Unfortunately, None of the 15 participating vehicles have ever completed more than 5% of the entire course. Therefore, a second DARPA Grand Challenge event was scheduled on October 8, 2005. Five of 23 vehicles successfully conquered the course, and Stanford's "Stanley" (see Figure 1.3) was crowned the winner with a result of 6 h 53 min (Thrun *et al.*, 2006). This robotic car was a milestone in the quest for the modern self-driving cars.


Figure 1.3: Stanley: The 2005 DARPA Grand Challenge Winner. (Thrun *et al.*, 2006)

Two years later, the "DARPA Urban Challenge" took place on November 3, 2007, and called for autonomous vehicles to drive 97 km through a mock urban environment in less than 6 hours, interacting with other moving vehicles and obstacles and obeying all traffic regulations. The vehicle "Boss" was declared the winner (Urmson *et al.*, 2008) and "Junior" won the second place (Montemerlo *et al.*, 2008). These vesicles were also regarded as the initial prototype of the Google self-driving car.

In a traditional programming scenario, a human programmer would use human understanding of the desired task and have to reason in advance to code a robot controller that is capable of responding to any situation the robot may face, no matter how unlikely. While such specialized programming is highly efficient, it is also expensive and limited to the situations the human operator had considered. If errors or new circumstances arise after the robot is deployed, the entire costly process may need to be repeated. While the mentioned DARPA

challenges were competed in unrehearsed courses, it is hard to imagine that all possible tasks can be preprogrammed. Therefore, robots need to be able to learn, either by themselves or with the help of supervision.

Motivated by the DARPA challenges and Google self-driving car, this dissertation concentrates on ensuring that robots can learn new skills and improve their existing abilities autonomously, and providing robots with the ability of making rational, intelligent decisions. By this means, robots can learn how to optimally adapt to uncertainty and unforeseen changes in order to tackle stochastic and dynamic environments.

## 1.2 Contributions of the Dissertation

This dissertation considers the intelligent control in autonomous navigation via robot learning from sensory inputs. The aim is for robot capabilities to be more easily extended and adapted to novel situations, even by users without programming ability.

We developed our research by investigating three major learning algorithms: reinforcement learning, learning from demonstration, and inverse reinforcement learning. The main contributions of this dissertation are summarized as follows:

1. Traditional reinforcement learning has a limitation of generalization of unvisited states, which also caught many researchers' attentions. Neural network, as a powerful supervised learning algorithm, can alleviate this problem due to its good generalization performance. Therefore, neural network was incorporated into reinforcement learning, and this method largely improved learning ability.

2. When we recall that how humans learn new things or skills during childhood, the word "imitation" comes to our mind. Robots should also acquire the basic instinct of imitating. Learning from demonstration, or imitation learning, is a good tool to enable robots to learn behaviors from expert demonstrations. We deduced a policy learning method that efficiently provided robots with a learning ability that can adapt to changing environments.

3. When experts could demonstrate optimal examples, learning from demonstrations showed sufficiently good performance. Most of the time, however, non-optimal examples are also provided. In such cases, learning from demonstrations may lead to poor performance. A better way is to learn the rewards in demonstrated states and then generalize to all undemonstrated states. Based on inverse reinforcement learning, or apprenticeship learning, we developed a method by introducing neural network, and we achieved a fast and robust learning algorithm.

All these three proposed methods were applied to mobile robot navigation experiments. From the results, we can safely conclude that our methods succeeded in endowing robots with learning abilities both by themselves via trial and error and by human demonstrations.

## 1.3 Organization of the Dissertation

This dissertation is organized as follows.

- In Chapter 2, we begin by formalizing the Markov Decision Processes (MDPs) and partially observable Markov decision processes (POMDPs) frameworks. We also review some standard algorithms for solving MDPs, known as dynamic programming. Then we presents some reinforcement learning algorithms that can be used in robotics.

- Chapter 3 begins by outlining the framework of learning from demonstration (LfD), from which we investigate its application on robotics. We present a method for learning stochastic policies from expert demonstrations and finding good controllers. Our method also applies well to autonomous navigation tasks, and uses a small amount of examples to learn from large and complicated sensor data. In real-world applications, examples cannot often be given by human experts, therefore, we give a modified version of A* pathfinding algorithm to simulate expert examples, which can generate paths much faster and is more adaptable to our problem.

- In Chapter 4, we investigate the self-learning ability of autonomous robots without expert demonstrations. We present a method under the reinforcement learning framework and use the artificial neural network to generalize and optimize the self-learning process. We successfully test our method in autonomous navigation tasks both in static and dynamic environments.

- In Chapter 5, we consider the potential damage to robots caused by failure during self-learning process, and we describe inverse reinforcement learning that robots learn rewards via interactions with the environment. We present our method neural inverse reinforcement learning, which incorporates neural network into inverse reinforcement learning and the robot learns a reward function as well as a policy.

- To conclude, Chapter 6 summaries the key points of our research and outlines avenues for further research.

# Chapter 2

# Markov Decision Processes and Reinforcement Learning in Robotics

## Contents

## 2.1 Introduction

In recent years, we have seen a fast development of using machine learning techniques onto robot control problems. Machine learning helps an agent to learn from example data or past experience to solve a given problem. In supervised learning, the learner is provided an explicit target for every single input, that is, the environment tells the learner what its response should be. In contrast, in reinforcement learning, only partial feedback is given to the learner about the learner's decisions. Therefore, under the framework of RL, the learner is a decision-making agent that takes actions in an environment and receives reward (or penalty) for its actions in trying to solve a problem. After a set of trial-and-error runs, it should learn the best policy, which is the sequence of actions that maximizes the total reward (Sutton & Barto, 1998).

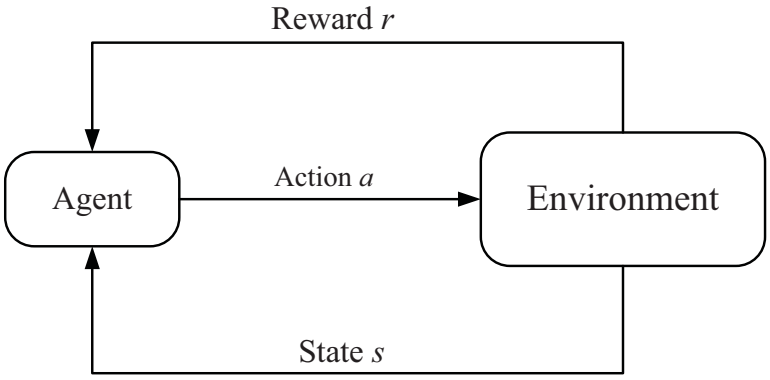



Figure 2.1: The mechanism of interaction between a learning agent and its environment in reinforcement learning.

Reinforcement learning is typically operated in a setting of interaction, shown in Figure 2.1: the learning agent interacts with an initially unknown environment, and receives a representation of the state and an immediate reward as the feedback. It then calculates an action, and subsequently undertakes it. This action causes the environment to transit into a new state. The agent receives the new representation and the corresponding reward, and the whole process repeats.

The environment in RL is typically formulated as a Markov Decision Process (MDP), and the goal is to learn to a control strategy so as to maximize the total reward which represents a long-term objective. In this chapter, we introduces the

structural background of Markov Decision Process and reinforcement learning in robotics.

## 2.2 Markov Decision Processes

A *Markov Decision Process* describes a sequential decision-making problem in which an agent must choose the sequence of actions that maximizes some reward-based optimization criterion (Puterman, 1994; Sutton & Barto, 1998). Formally, an MDP is a tuple $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, r, \gamma\}$, where

- $\mathcal{S} = \{s_1, \ldots, s_N\}$ is a finite set of $N$ *states* that represents the dynamic environment,
- $\mathcal{A} = \{a_1, \ldots, a_k\}$ is a set of $k$ *actions* that could be executed by an agent,
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \longmapsto [0, 1]$ is a *transition probability function*, or *transition model*, where $\mathcal{T}(s, a, s')$ stands for the state transition probability upon applying action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$ leading to state in state $s' \in \mathcal{S}$, i.e. $\mathcal{T}(s, a, s') = P(s' \,|\, s, a)$,
- $r : \mathcal{S} \times \mathcal{A} \longmapsto \mathbb{R}$ is a *reward function* with absolute value bounded by $R_{max}$; $r(s, a)$ denotes the immediate reward incurred when action $a \in \mathcal{A}$ is executed in state $s \in \mathcal{S}$,
- $\gamma \in [0, 1)$ is a *discount factor*.

Given an MDP $\mathcal{M}$, the agent-environment interaction in Figure 2.1 happens as follows: let $t \in \mathbb{N}$ denote the current time, let $S_t \in \mathcal{S}$ and $A_t \in \mathcal{A}$ denote the random state of the environment and the action chosen by the agent at time $t$, respectively. Once the action is selected, it is sent to the system, which makes a transition:

$$(S_{t+1}, R_{t+1}) \sim P(\cdot \,|\, S_t, A_t). \tag{2.1}$$

In particular, $S_{t+1}$ is random and $P(S_{t+1} = s' \,|\, S_t = s, A_t = a) = \mathcal{T}(s, a, s')$ holds true for any $s, s' \in \mathcal{S}, a \in \mathcal{A}$. Furthermore, $\mathbb{E}\left[R_{t+1} \,|\, S_t, A_t\right] = r(S_t, A_t)$. The agent then observes the next state $S_{t+1}$ and reward $R_{t+1}$, chooses a new action $A_{t+1} \in \mathcal{A}$ and the process is repeated.

The Markovian assumption (Sutton & Barto, 1998) implies that the sequence of state-action pairs specifies the transition model $\mathcal{T}$:

$$P(S_{t+1} \mid S_t, A_t, \cdots, S_0, A_0) = P(S_{t+1} \mid S_t, A_t). \tag{2.2}$$

State transitions can be deterministic or stochastic. In the deterministic case, taking a given action in a given state always results in the same next state; while in the stochastic case, the next state is a random variable.

The goal of the learning agent is to figure out a theory of choosing the actions so as to maximize the expected total discounted reward:

$$\mathcal{R} = \sum_{t=0}^{\infty} \gamma^t R_{t+1}. \tag{2.3}$$

If $\gamma < 1$ then the rewards received far in the future are exponentially less worthy than those received at the first stage.

### 2.2.1 Policies and Value Functions

The agent selects its actions according to a special function called *policy*. A policy is defined as a mapping $\pi : \mathcal{S} \times \mathcal{A} \longmapsto [0, 1]$ that assigns to each $s \in \mathcal{S}$ a distribution $\pi(s, \cdot)$ over $\mathcal{A}$, satisfying $\sum_{a \in \mathcal{A}} \pi(a \mid s) = 1, \forall s \in \mathcal{S}$.

A *deterministic stationary policy* is the case that for all $s \in \mathcal{S}$, $\pi(\cdot \mid s)$ is concentrated on a single action, i.e. at any time $t \in \mathbb{N}, A_t = \pi(S_t)$. A *stochastic stationary policy* is a function that maps each state into a probability distribution over the different possible actions, i.e., $A_t \sim \pi(\cdot \mid S_t)$. The class of all stochastic stationary policies is denoted by $\Pi$.

Application of a policy is done in the following way. First, a start state $S_0$ is generated. Then, the policy $\pi$ suggests the action $A_0 = \pi(S_0)$ and this action is performed. Based on the transition function $\mathcal{T}$ and reward function $r$, a transition is made to state $S_1$, with a probability $\mathcal{T}(S_0, A_0, S_1)$ and a reward $R_1 = r(S_0, A_0, S_1)$ is received. This process continues, producing a sequence $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, ...$, as shown in Figure 2.2.

*Value functions* are functions of states (or of state-action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of "how good" here is defined in terms

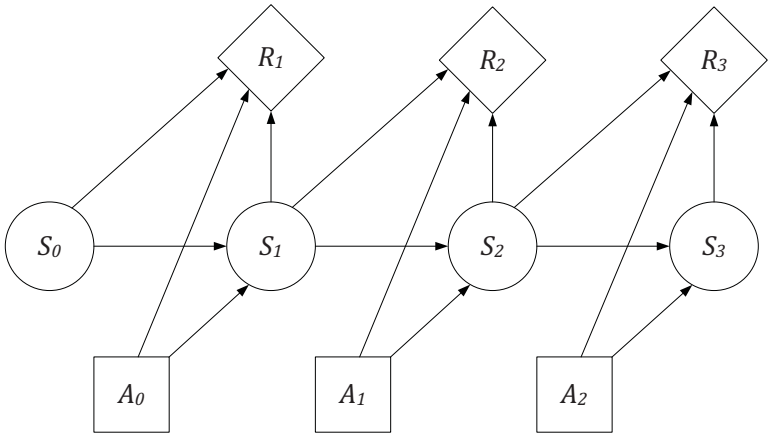



Figure 2.2: Decision network of a finite MDP.

of future rewards that can be expected, or, to be precise, in terms of expected return. Of course the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular policies (Sutton & Barto, 1998).

Given a a policy $\pi$, the value function is defined as a function $V^\pi : \mathcal{S} \longmapsto \mathbb{R}$ that associates to each state the expected sum of rewards that the agent will receive if it starts executing policy $\pi$ from that state:

$$V^\pi(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t r(S_t, A_t) \,|\, S_0 = s \right], \qquad \forall s \in \mathcal{S}. \tag{2.4}$$

$S_t$ is the random variable representing the state at time $t$, $A_t$ is the random variable corresponding to the action taken at that time instant and is such that $P(A_t = a \,|\, S_t = s) = \pi(s, a)$. $(S_t, A_t)_{t \geq 0}$ is the sequence of random state-action pairs generated by executing the policy $\pi$.

The value function of a stationary policy can also be recursively defined as:

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^{t} r(S_t, A_t) \,|\, S_0 = s\right] \\ &= \mathbb{E}_{\pi}\left[r(S_0, A_0) + \sum_{t=1}^{\infty}\gamma^{t} r(S_t, A_t) \,|\, S_0 = s\right] \\ &= r(s, \pi(s)) + \mathbb{E}_{\pi}\left[\sum_{t=1}^{\infty}\gamma^{t} r(S_t, A_t) \,|\, S_0 = s\right] \\ &= r(s, \pi(s)) + \gamma\mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^{t} r(S_t, A_t) \,|\, S_0 \sim \mathcal{T}(s, \pi(s), \cdot)\right] \\ &= r(s, \pi(s)) + \gamma\sum_{s'\in\mathcal{S}}\mathcal{T}(s, \pi(s), s')V^{\pi}(s'), \end{aligned} \tag{2.5}$$

where $\pi(s)$ is the action associated to state $s$.

If the uncertainty of a stochastic policy $\pi(s)$ is taken into account, $V^{\pi}(s)$ can also be specifically written as:

$$V^{\pi}(s) = \sum_{a\in\mathcal{A}(s)}\pi(s, a)\left(r(s, a) + \gamma\sum_{s'\in\mathcal{S}}\mathcal{T}(s, a, s')V^{\pi}(s')\right). \tag{2.6}$$

Similarly, the *action-value function* $Q^{\pi} : \mathcal{S}\times\mathcal{A} \longmapsto \mathbb{R}$ underlying a policy $\pi$ is defined as

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[\sum_{t=0}^{\infty}\gamma^{t} r(S_t, A_t) \,|\, S_0 = s, A_0 = a\right], \tag{2.7}$$

where $S_t$ is distributed according to $\pi(S_t, \cdot)$ for all $t > 0$. Finally, we defined the *advantage function* associated with $\pi$ as

$$A^{\pi} = Q^{\pi}(s, a) - V^{\pi}(s). \tag{2.8}$$

A policy that maximizes the expected total discounted reward over all states is called an *optimal policy*, denoted $\pi^*$. For any finite MDP, there is at least one optimal policy.

The *optimal value function* $V^*$ and the *optimal action-value function* $Q^*$ are

defined by

$$\begin{aligned} V^*(s) &= \sup_\pi V^\pi(s), \qquad s \in \mathcal{S}, \\ Q^*(s,a) &= \sup_\pi Q^\pi(s,a), \qquad s \in \mathcal{S}, a \in \mathcal{A}. \end{aligned} \tag{2.9}$$

Moreover, the optimal value- and action-value functions are connected by the following equations:

$$V^*(s) = \sup_{a \in \mathcal{A}} Q^*(s,a), \qquad s \in \mathcal{S}, \tag{2.10}$$

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \,|\, s,a) V^*(s'), \qquad s \in \mathcal{S}, a \in \mathcal{A}. \tag{2.11}$$

It turns out that $V^*$ and $Q^*$ satisfy the so-called Bellman optimality equations (Puterman, 1994). In particular,

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \,|\, s,a) \max_{b \in \mathcal{A}} Q^*(s',b), \tag{2.12}$$

$$V^*(s) = \max_{a \in \mathcal{A}} r(s,a) + V^*(s'). \tag{2.13}$$

We call a policy that satisfies $\sum_{a \in \mathcal{A}} \pi(a \,|\, s) Q(s,a) = \max_{a \in \mathcal{A}} Q(s,a)$ at all states $s \in \mathcal{S}$ *greedy* w.r.t. the function Q. It is known that all policies that are greedy w.r.t. $Q^*$ are optimal and all stationary optimal policies can be obtained these way.

Here, we present the following important results concerning MDP (Sutton & Barto, 1998):

**Theorem 2.1** *(Bellman Equations). Let a Markov Decision Problem* $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, r, \gamma\}$ *and a policy* $\pi : \mathcal{S} \times \mathcal{A} \longrightarrow [0,1]$ *be given. Then,* $\forall s \in \mathcal{S}, a \in \mathcal{A}$, $V^\pi$ *and* $Q^\pi$ *satisfy*

$$V^\pi(s) = r(s,\pi(s)) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s,\pi(s),s') V^\pi(s'), \tag{2.14}$$

$$Q^\pi(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s') V^\pi(s'). \tag{2.15}$$

**Theorem 2.2** *(Bellman Optimality). Let a Markov Decision Problem* $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, r, \gamma\}$ *and a policy* $\pi : \mathcal{S} \times \mathcal{A} \longrightarrow [0,1]$ *be given. Then,* $\pi$ *is an optimal*

*policy for* $\mathcal{M}$ *if and only if,* $\forall s \in \mathcal{S}$,

$$\pi(s) \in \arg\max_{a \in \mathcal{A}} Q^{\pi}(s, a). \tag{2.16}$$

The transition probability $\mathcal{T}(s, a, s') = P(s' \mid s, a)$.

### 2.2.2 Partially Observable Markov Decision Processes

In the setting of an MDP, the agent should obtain a precise state information of the environment at any moment. Unfortunately, this assumption may not hold in practice in respect that the agent observes the world through limited and imperfect receptors, and the information contained in the perceptions is not sufficient for determining the state. *Partially Observable Markov Decision Processes* (POMDPs) (Smallwood & Sondik, 1973) was introduced to handle such cases.

Formally, a POMDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{O}, Z, R \rangle$ where $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, R \rangle$ is an MDP, and:

- $\mathcal{O}$ is a set of observations (perceptions),
- $Z$ is an observation function: $Z(o, s, a)$ is the probability of observing $o \in \mathcal{O}$ if the state of the system is $s$ and the action that led to this state is $a$.

The observations can be aliased (the same observation may be observed in different states) and stochastic (different observations may be observed in the same state). Consequently, the state of the system cannot be determined from the observations. Instead, an observation can be seen as an evidence about the state. The agent's belief about the hidden state is a probability distribution on all the possible states, called *the belief state*:

$$b_t = \left[ Pr(s_t = x^0), Pr(s_t = x^1), \ldots, Pr(s_t = s^{|\mathcal{S}|-1}) \right]^{\top} \tag{2.17}$$

The agent starts with an initial belief state $b_0$, and every time an action $a_t$ is executed and an observation $o_{t+1}$ is received, the belief state $b_t$ is updated by using Bayes' Rule:

$$
\begin{aligned}
b_{t+1}(x) &= Pr(s_{t+1} = s \,|\, b_t, a_t, o_{t+1}) \\
&= \frac{Pr(s_{t+1} = s, o_{t+1} \,|\, b_t, a_t)}{Pr(o_{t+1} \,|\, b_t, a_t)} \\
&= \frac{\sum_{s' \in \mathcal{S}} b_t(s') \mathcal{T}(s, a_t, s') Z(o_{t+1}, s, a_t)}{\sum_{s' \in \mathcal{S}} \sum_{s'' \in \mathcal{S}} b_t(s') \mathcal{T}(s', a_t, s'') Z(o_{t+1}, s'', a_t)}
\end{aligned} \tag{2.18}
$$

Therefore, the belief state $b_{t+1}$ is a nonlinear function of the previous belief state $b_t$, action $a_t$ and observation $o_{t+1}$:

$$
b_{t+1} = \tau(b_t, a_t, o_{t+1}) \tag{2.19}
$$

The belief state $b_t$ at time instant $t$ allows us to calculate the probability of any observation $o$ at time instant $t + 1$ as:

$$
Pr(o_{t+1} = o \,|\, b_t, a_t) = \sum_{s \in \mathcal{S}} \sum_{s' \in \mathcal{S}} b_t(s) \mathcal{T}(s, a_t, s') Z(o, s', a_t) \tag{2.20}
$$

The expected reward of executing action $a$ for a belief state $b_t$ is given by:

$$
r(a \,|\, b_t) = \sum_{s \in \mathcal{S}} b_t(s) R(s, a) \tag{2.21}
$$

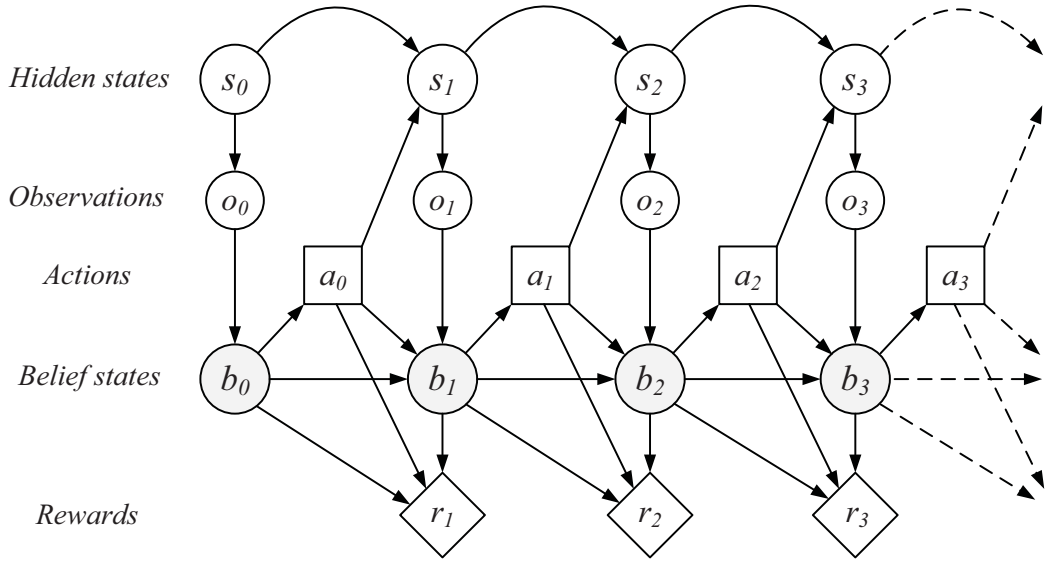



Figure 2.3: Schema of a Partially Observable Markov Decision Process.

A Markovian belief state allows a POMDP to be formulated as a Markov decision process where every belief is a state. The resulting *Belief Markov Decision Process* will thus be defined on a continuous state space, since there are infinite beliefs for any given POMDP (Kaelbling *et al.*, 1998). The belief MDP is defined as a tuple $\langle \mathcal{B}, \mathcal{A}, \tau, r, \gamma \rangle$ where:

- $\mathcal{B}$ is the set of belief states over the POMDP states,
- $\mathcal{A}$ is the same set of actions as for the original POMDP,
- $\tau$ is the belief state transition function,
- $r : \mathcal{B} \times \mathcal{A} \longrightarrow \mathbb{R}$ is the reward function on belief states,
- $\gamma$ is the discount factor equal to the $\gamma$ in the original POMDP.

In belief MDP, $\tau$ and $r$ need to be derived from the original POMDP. For all $b, b' \in \mathcal{B}, a \in \mathcal{A}$:

$$\tau(b, a, b') = \sum_{o \in \mathcal{O}} Pr(b' \mid b, a, o) Pr(o \mid a, b), \tag{2.22}$$

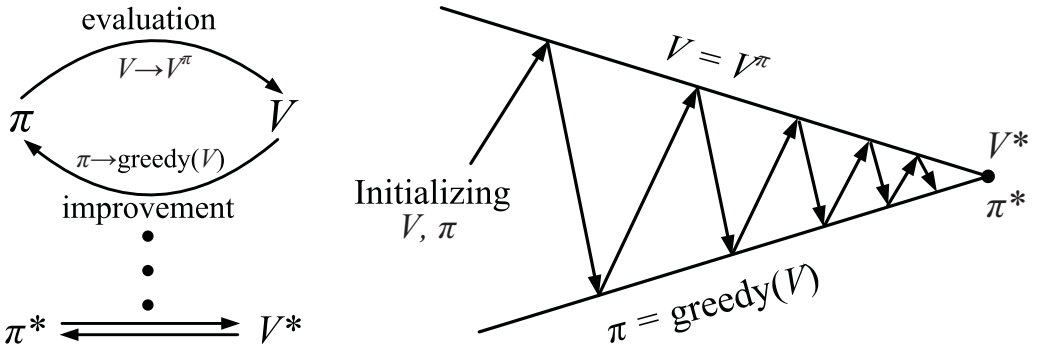
$$r(b, a) = \sum_{x \in \mathcal{S}} b(s) R(s, a). \tag{2.23}$$

Figure 2.3 illustrates the temporal series of belief states. The Markov property implies that the belief state and the full history of the system (sequence of actions and observations) contain exactly the same information about the current state, and consequently, about all the future events.

## 2.3 Dynamic Programming: Model-Based Algorithms

Dynamic programming (DP) is a method for computing an optimal policy $\pi^*$ in order to solve a given Markov decision process.

Dynamic programming assumes full knowledge of the Markov decision process, including the transition dynamics of the environment and the reward function (Bertsekas, 1996). Therefore, they are classified into *model-based* learning




(a) Interaction of policy evaluation and improvement processes

(b) The convergence of both the value function and the policy to their optimals

Figure 2.4: Generalized Policy Iteration (Sutton & Barto, 1998)

algorithms. On the contrary are *model-free* learning algorithms, which do not require a perfect model of the environment, and will be introduced them later in this chapter.

Dynamic programming algorithms for solving MDPs can be categorized into one of the two families: value iteration (VI) and policy iteration (PI) (Sutton & Barto, 1998). Both of these approaches share a common underlying mechanism, the *generalized policy iteration* (GPI) principle (Sutton & Barto, 1998), depicted in Figure 2.4. This principle consists of two interaction processes. The first step, *policy evaluation*, estimates the utility of the current policy $\pi$, that is, it computes the value $V^\pi$. This step gathers information about the policy for computing the second step, the *policy improvement* step. In this step, the values of the actions are evaluated for every state, in order ot find possible improvements, that is, possibly other actions in particular states that are better than the action the current policy proposes. This step computes an improved policy $\pi'$ from the current policy $\pi$ using the information in $V^\pi$. As long as both processes continue to update all states, the ultimate goal is to converge to the optimal value function and an optimal policy. Figure 2.4(b) presents a geometric metaphor for convergence of both the value function and the policy in GPI.

### 2.3.1 Policy Iteration

*Policy iteration* iterates between the two processes of GPI. This is repeated until converging to an optimal policy. This method is depicted in Algorithm 1.

**Algorithm 1** Policy Iteration (Sutton & Barto, 1998)

**Input:** An MDP model $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, r, \gamma \rangle$;

`/* Initialization */`

1: $t = 0, k = 0$;
2: $\forall s \in \mathcal{S}$: Initialize $\pi_t(s)$ with an arbitrary action;
3: $\forall s \in \mathcal{S}$: Initialize $V_k(s)$ with an arbitrary value;
4: **repeat**

`/* Policy evaluation */`

5: **repeat**
6: $\forall s \in \mathcal{S} : V_{k+1}(s) = r(s, \pi_t(s)) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s, \pi_t(s), s') V_k(s')$;
7: $k \leftarrow k + 1$;
8: **until** $\forall s \in \mathcal{S} : |V_k(s) - V_{k-1}(s)| < \epsilon$;

`/* Policy improvement */`

9: $\forall s \in \mathcal{S} : \pi_{t+1}(s) = \arg\max_{a \in \mathcal{A}} \left[ r(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') V_k(s') \right]$;
10: $t \leftarrow t + 1$;
11: **until** $\pi_t = \pi_{t-1}$;
12: $\pi^* = \pi_t$;

**Output:** An optimal policy $\pi^*$.

It consists in starting with a randomly chosen policy $\pi_t$ and a random initialization of the corresponding value function $V_k$, for $k = 0$ and $t = 0$ (Steps 1 to 3), and iteratively repeating the *policy evaluation* and the *policy improvement* operations.

Policy evaluation (Steps 5 to 8) consists in calculating the action value of policy $\pi_{t+1}$ by solving the solving the equation (2.15) for all the states $s \in \mathcal{S}$. An efficient iterative way to solve this equation is to initialize the value function of $\pi_{t+1}$ with the value function $V_k$ of the previous policy, and then repeat the operation:

$$\forall s \in \mathcal{S} : V_{k+1}(s) = r(s, \pi_t(s)) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s, \pi_t(s), s') V_k(s'), \tag{2.24}$$

until $\forall s \in \mathcal{S} : |V_k(s) - V_{k-1}(s)| < \epsilon$, for a predefined error threshold $\epsilon$.

Policy improvement (Steps 9 to 10) consists in finding the greedy policy $\pi_{t+1}$

given the value function $V_k$:

$$\forall s \in \mathcal{S} : \pi_{t+1}(s) = \arg\max_{a \in \mathcal{A}} \left[ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s') V_k(s') \right]. \tag{2.25}$$

This process stops when $\pi_t = \pi_{t-1}$, in which case $\pi_t$ is an optimal policy, i.e., $\pi^* = \pi_t$.

In sum, PI generates a direct sequence of alternating policies and value functions:

$$\pi_0 \to V^{\pi_0} \to \pi_1 \to V^{\pi_1} \to \cdots \to \pi^* \to V^* \to \pi^*$$

The policy evaluation processes occur in the transitions of $\pi_t \to V^{\pi_t}$; while the $V^{\pi_t} \to \pi_{t+1}$ conversions are realized by the policy improvement processes.

### 2.3.2 Value Iteration

The main drawback of policy iteration is that a complete policy evaluation is involved in each iteration. Value iteration consists in overlapping the evaluation and improvement processes.

Instead of completely separating the evaluation and improvement processes, the *value iteration* approach breaks off evaluation after just one iteration. In fact, it immediately blends the policy improvement step into its iterations, thereby purely focusing on estimating directly the value function.

Value iteration, described in Algorithm 2, can be written as a simple backup operation:

$$\forall s \in \mathcal{S} : V_{k+1}(s) = \max_{a \in \mathcal{A}} \left[ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s') V_k(s') \right]. \tag{2.26}$$

This operation is repeated (Steps 3 to 6) until $\forall s \in \mathcal{S} : |V_k(s) - V_{k-1}(s)| < \epsilon$, in which case the optimal policy is simply the greedy policy with respect to the value function $V_k$ (Step 7).

VI produces the following sequence of value functions:

$$V_0 \to V_1 \to V_2 \to V_3 \to V_4 \to V_5 \to \cdots \to \pi^*$$

**Algorithm 2** Value Iteration (Sutton & Barto, 1998)

**Input:** An MDP model $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, r, \gamma \rangle$;
1: $k = 0$;
2: $\forall s \in \mathcal{S}$: Initialize $V_k(s)$ with an arbitrary value;
3: **repeat**
4: $\quad \forall s \in \mathcal{S} : V_{k+1}(s) = \max_{a \in \mathcal{A}} \left[ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s') V_k(s') \right]$;
5: $\quad k \leftarrow k + 1$;
6: **until** $\forall s \in \mathcal{S} : |V_k(s) - V_{k-1}(s)| < \epsilon$;
7: $\forall s \in \mathcal{S} : \pi^*(s) = \arg\max_{a \in \mathcal{A}} \left[ r(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s') V_k(s') \right]$;
**Output:** An optimal policy $\pi^*$.

## 2.4 Reinforcement Learning: Model-Free Algorithms

Reinforcement learning (RL) is a machine learning framework for solving sequential decision problems that can be modeled as MDPs (Kaelbling *et al.*, 1996). Unlike dynamic programming that assumes the availability of a perfect model of the environment, RL is primarily concerned with how to obtain an optimal policy when such a model is not available. Therefore, reinforcement learning is model-free. In addition, RL adds to MDPs a focus on approximation and incomplete information, and the need for sampling and exploration to gather statistical knowledge about this unknown model.

In a RL problem, the agent and its environment may be modeled being in a state $s \in S$ and can perform actions $a \in A$, each of which may be members of either discrete or continuous sets and can be multi-dimensional. A state $s$ contains all relevant information about the current situation to predict future states. An action $a$ is used to control the state of the system. For every step, the agent also gets a reward $R$, which is a scalar value and assumed to be a function of the state and observation. It may equally be modeled as a random variable that depends on only these variables. In the navigation task, a possible reward could be designed based on the energy costs for taken actions and rewards for reaching targets. Reinforcement learning is designed to find a policy $\pi$ from states to actions, that picks action $a$ in given state $s$ maximizing the cumulative expected reward. The policy $\pi$ is either deterministic or stochastic. The former always uses the exact same action for a given state in the form $a = \pi(s)$, the later draws a sample from a distribution over actions when it encounters a state, i.e., $a \sim \pi(s,a) = P(a|s)$. The reinforcement learning agent needs to discover

the relations between states, actions, and rewards. Hence exploration is required which can either be directly embedded in the policy or performed separately and only as part of the learning process. Different types of reward functions are commonly used, including rewards depending only on the current state $R = R(s)$, rewards depending on the current state and action $R = R(s, a)$, and rewards including the transitions $R = R(s', a, s)$.

A detailed survey of reinforcement learning in robotics can be found in (Kober *et al.*, 2013).

### 2.4.1 Goals of Reinforcement Learning

The goal of reinforcement learning is to discover an optimal policy $\pi^*$ that maps states or observations to actions so as to maximize the expected return $J$, which corresponds to the cumulative expected reward. A finite-horizon model only attempts to maximize the expected reward for the horizon $H$, i.e., the next $H$ (time-)steps $h$:

$$J = \mathbb{E}\left\{\sum_{h=0}^{H} R_h\right\}. \tag{2.27}$$

This setting can also be applied to model problems where it is known how many steps are remaining.

Alternatively, future rewards can be discounted by a discount factor $\gamma$ (with $0 \leq \gamma < 1$):

$$J = \mathbb{E}\left\{\sum_{h=0}^{\infty} \gamma^h R_h\right\}. \tag{2.28}$$

Two natural goals arise for the learner. In the first, we attempt to find an optimal strategy at the end of a phase of training or interaction. In the second, the goal is to maximize the reward over the whole time the robot is interacting with the world.

In contrast to supervised learning, the learner must first discover its environment and is not told the optimal action it needs to take. To gain information about the rewards and the behavior of the system, the agent needs to explore by considering previously unused actions or actions it is uncertain about. It needs to decide whether to play it safe and stick to well known actions with

(moderately) high rewards or to dare trying new things in order to discover new strategies with an even higher reward. This problem is commonly known as the *exploration-exploitation trade-off*.

RL relies on the interaction between a learning agent and its environment (see Figure 2.1), the process is similar:

1. A learning agent interacts with its environment in discrete time steps;
2. At each time step $t$, the agent observes the environment, and receives a representation of state $s_t$ and a reward $r_t$;
3. The agent infers an action $a_t$, and subsequently undertaken in the environment.
4. The agent observes the new environment, and receives a new state representation $s_{t+1}$ and an associated reward $r_{t+1}$.

Based on how the agent chooses an action, RL can be distinguished between off-policy and on-policy methods. *Off-policy* algorithms learn independent of the employed policy, i.e., an explorative strategy that is different from the desired final policy can be employed during the learning process. *On-policy* algorithms collect sample information about the environment using the current policy. As a result, exploration must be built into the policy and determines the speed of the policy improvements. Such exploration and the performance of the policy can result in an exploration-exploitation trade-off between long- and short-term improvement of the policy. A simple exploration scheme known as $\epsilon$-greedy, performs a random action with probability $\epsilon$ and otherwise greedily follows the state-action values.

### 2.4.2 Monte Carlo Methods

*Monte Carlo methods* use sampling in order to estimate the value function and discover the optimal policy (Sutton & Barto, 1998). This procedure can be used to replace the policy evaluation step of the dynamic programming-base methods above. Unlike DP, Monte Carlo methods do not assume complete knowledge of the environment. Monte Carlo methods are *model-free*, i.e., they do not need an explicit transition function. They require only experience – sample sequences

of states, actions, and rewards from online or simulated interaction with an environment. Learning from online experience requires no prior knowledge of the environment's dynamics, yet can still attain optimal behavior. Learning from simulated experience requires a model, but the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required by dynamic programming methods.

Monte Carlo methods solve reinforcement learning problems based on averaging sample returns. They perform rollouts by executing the current policy on the system, hence operating on-policy. The frequencies of transitions and rewards are kept track of and used to form estimates of the value function. For example, in an episodic setting the state-action value of a given state action pair can be estimated by averaging all the returns that were received when starting from them.

### 2.4.3 Temporal Difference Methods

*Temporal Difference (TD) Methods* is a combination of Monte Carlo methods and dynamic programming methods (Sutton & Barto, 1998). Unlike Monte Carlo methods, TD learning methods do not have to wait until an estimate of the return is available (i.e., at the end of an episode) to update the value function. Instead, they use temporal errors and only have to wait until the next time step. The temporal error is the difference between the old estimate and a new estimate of the value function, taking into account the reward received in the current sample. These updates are done iteratively and, in contrast to dynamic programming methods, only take into account the sampled successor states rather than the complete distributions over successor states. Like the Monte Carlo methods, these methods are model-free, as they do not use a model of the transition function to determine the value function, and can learn directly from raw experience without a model of the environment's dynamics. In this setting, the value function cannot be calculated analytically but has to be estimated from sampled transitions in the MDP.

Q-Learning (Watkins, 1989) is a representative off-policy, model-free RL algorithm. It incrementally processes the transition samples. Q-value is updated iteratively by

$$Q'(s,a) \leftarrow Q(s,a) + \alpha \left( r(s,a) + \gamma \max_{b \in \mathcal{A}} Q(s',b) - Q(s,a) \right). \tag{2.29}$$

SARSA (Rummery & Niranjan, 1994) is a representative on-policy, model-free RL algorithm. Different from Q-learning that uses $\max_{b \in \mathcal{A}} Q(s',b)$ for estimating future rewards, SARSA uses $Q(s',a')$ for $a'$ the action executed in $s'$ under the current policy that generates the transition sample $(s,a,,r,s',a')$. Mathematically, the update rule is:

$$Q'(s,a) \leftarrow Q(s,a) + \alpha \left( r(s,a) + \gamma Q(s',a') - Q(s,a) \right). \tag{2.30}$$

If each action is executed in each state an infinite number of times, and for all state-action pairs $(s,a)$, the learning rate $\alpha$ is decayed appropriately, the Q-values will converge with probability 1 to the optimal $Q^*$ (Watkins & Dayan, 1992). Similar guarantee of convergence for SARSA can be found in (Singh *et al.*, 2000) with a more strict requirement on the exploration of all states and actions.

More contents about reinforcement learning will be subsequently presented in Chapter 4.

## 2.5 Mobile Robot Model

Figure 2.5: Robotino®: a mobile robot system for education and research.

The mobile robot model used in this dissertation is built according to the mobile robot system Robotino® from Festo Didactic*, as shown in Figure 2.5. It is based on an omnidirectional drive assembly, which enables the system to roam freely. The Robotino® is equipped with a total of nine infrared sensors arranged around its base at an angle of 40° to each other, and each distance sensor reads out a voltage level whose value depends on the distance to a reflective object.

Therefore, our mobile robot model is assumed to have three wheels, one in front and two in back. The robot is equipped with nine distance sensors to determine the distances of objects within its surroundings. The sensors are arranged at an angle of 40° to each other, and each sensor can cover a scope of detection of 40°, as shown in Figure 2.6. $(S_i)_{\{1\leq i\leq 9\}}$ are the nine sensors and $(R_i)_{\{1\leq i\leq 9\}}$ representing the corresponding scopes of detection.

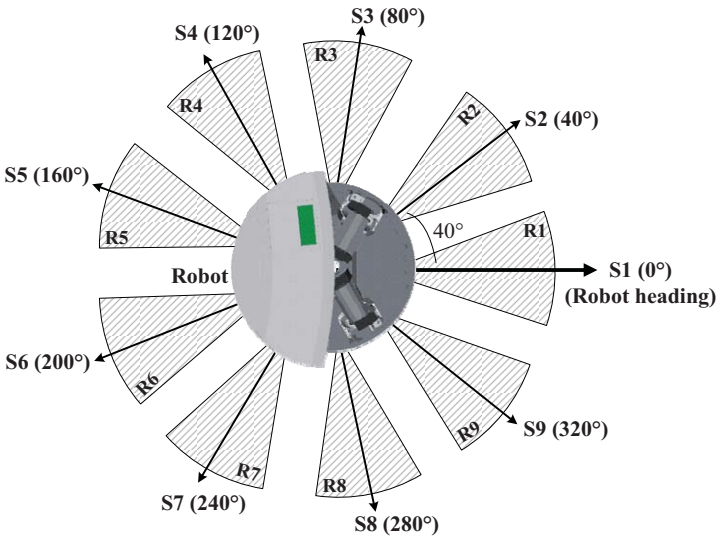



Figure 2.6: The mobile robot model with nine sensors and their corresponding sensing areas.

The robot navigation environment consists of its target and the obstacles, as shown in Figure 2.7. The initial and target positions are predefined to the robot, and the robot will try to reach the target in a collision-free path in spite of the presence of static or moving obstacles.

$d_{r-t}$ is the distance between the robot and the target, and $d_{r-o}$ is the distance between the robot and the obstacle.

*http://www.festo-didactic.com/int-en/services/robotino/

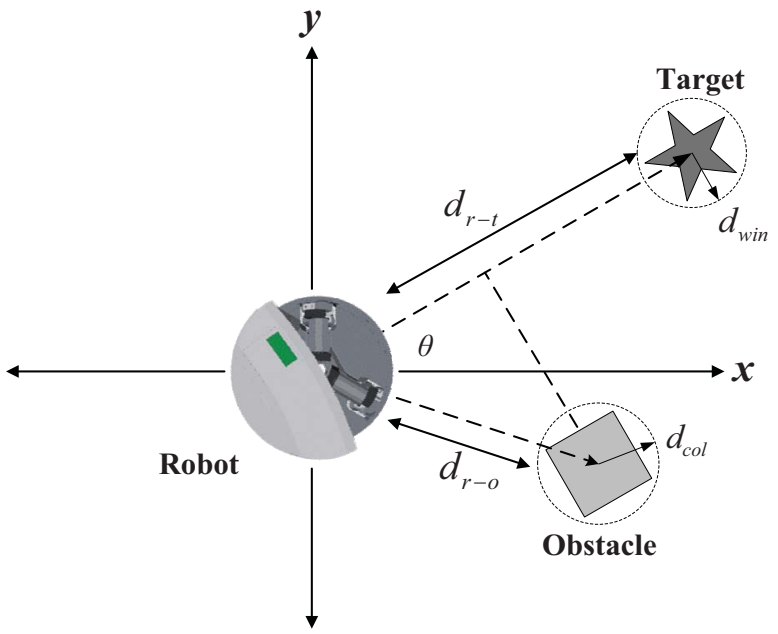



Figure 2.7: The mobile robot environment with a target and an obstacle.

### 2.5.1 Uncertainty in Mobile Robots

For mobile robots, uncertainty is everywhere. Wheels may slip, sensors may be affected by noise, and obstacles move unpredictably. Truly autonomous robots or decision-making agents must act in ways that are robust to these sorts of failures and unexpected events which we may think of in general as uncertainty.

Uncertainty can take many forms, but for brevity and clarity we place our attention to two important types (O'kane *et al.*, 2005):

- **Prediction uncertainty** occurs when the effects of actions are not fully predictable. This can be thought of as an uncertainty in future states.
- **Sensing uncertainty** is the uncertainty in the current state. This occurs, for example, in robots that have limited or imperfect sensing. We also admit the case where robots have no sensing at all.

The uncertainty in mobile robots may result in the fact that when a mobile robot tries to execute an action in a given state of environment, the action does not always lead to the expected result, because the information represented by the state may not precisely determine the outcome of the actions.

## 2.6 Conclusion

This chapter has presented the Markov decision processes, which are the underlying structure of robot learning methods presented in this dissertation. Several classical algorithms for solving MDPs were also briefly introduced. The fundamental concepts of the reinforcement learning was then brought. Finally, we describe the mobile robot model that is used throughout this dissertation.

# Chapter 3

# Policy Learning from Multiple Demonstrations

## Contents

## 3.1 Introduction

Nowadays, robots are designed to perform complicated tasks, such as grasping and manipulating objects, navigating in outdoor environments, and driving in urban roads. Traditionally, robots had to be tediously hand programmed for every task they performed, which is a laborious and time-intensive engineering process (Tian *et al.*, 2015). Rather than manually pre-programming each desired behavior, an appropriate robot controller is always preferable that can be derived from observations of a human's own performance. In this context, learning mechanisms are needed to acquire knowledge from such an interaction process.

Learning from demonstration (LfD) is a machine learning mechanism that enables robotic systems to autonomously perform new tasks, and LfD is inspired by the way humans learn from being guided by experts, from infancy through adulthood. The main principle of robot learning from demonstration is that end-users can teach robots new tasks without programming (Billard & Grollman, 2013).

The basic idea is that the robot learns a behavior from one or several demonstrations performed by a teacher. Within LfD, a mapping between world states and actions is learned from examples, or demonstrations. One demonstration is defined as a sequence of state-action pairs that are recorded during the teacher's demonstration of the desired robot behavior. LfD algorithms utilize this dataset of examples to drive a policy that reproduces the demonstrated behavior. However, we also need the system to adapt to new and unseen conditions. Therefore, besides learning, generalizing is another key point to LfD, especially to tackle the curse of dimensionality in high dimensional spaces.

We proposed our method of policy learning from multiple demonstrations (Xia *et al.*, 2015) by incorporating artificial neural network (ANN) into learning from demonstration due to its properties such as nonlinear mapping, ability to learn from examples, good generalization performance, and capability to approximate

an arbitrary function given sufficient number of neurons. Different from direct imitation learning which could only reproduce what is seen in the demonstrations, our method can hence generalize to undemonstrated states and endow robots with the ability to learn what it means to perform a task by generalizing from observing multiple demonstrations.

As we mentioned before, the navigating capability is quite important for mobile robots. However, in some hazardous cases, providing reliable demonstrations places a burden on the human expert, even impossible. An alternative way is to provide robots with computer-based examples and let robots observe and learn. Based on this assumption, we also proposed an modified version of A* pathfinding algorithm to play the part of expert demonstrations.

## 3.2 Related Work

Learning from demonstration (Argall *et al.*, 2009) can be a powerful and natural tool for developing robot control policies. Many techniques rely upon an unknown mapping from teacher observations and actions to robot observations and actions (Alissandrakis *et al.*, 2002; Robins *et al.*, 2004; Ude *et al.*, 2004). A wide variety of demonstration training methods have been explored in previous work, including teleoperation (Saunders *et al.*, 2006; van Lent & Laird, 2001), direct manipulation of the learning agent (Atkeson & Schaal, 1997), object grasping (Sweeney & Grupen, 2007), the grocery checkout task (Jain *et al.*, 2013), etc. In Abbeel *et al.* (2010), acrobatic trajectories of an helicopter are learned by recording the tilt and pan motion of the helicopter when teleoperated by an expert pilot.

Nicolescu & Matarić (2001, 2003) present a learning framework based on demonstration, generalization and teacher feedback, in which training is performed by having the robot follow a human and observe its actions. A high-level task representation is then constructed by analyzing the experience with respect to the robot's underlying capabilities. The authors also describe a generalization of the framework that allows the robot to interactively request help from a human in order to resolve problems and unexpected situations. This interaction is implicit as the agent has no direct method of communication; instead, it attempts to convey its intentions by communicating though its actions.

Lockerd & Breazeal (2004) and Breazeal *et al.* (2004) demonstrate a robotic system where high-level tasks are taught through social interaction. In this framework, the teacher interacts with the agent through speech and visual inputs, and the learning agent expresses its internal state through emotive cues such as facial and body expressions to help guide the teaching process. The outcome of the learning is a goal-oriented hierarchical task model. In later work (Thomaz & Breazeal, 2008), the authors examine ways in which people give feedback when engaged in an interactive teaching task. Although the study's focus is to examine the use of a human-controlled reward signal in reinforcement learning, the authors also find that users express a desire to guide or control the agent while teaching. This result supports our belief that, for many robotic domains, teleoperation provides an easy and intuitive human-robot communication method.

Bentivegna *et al.* (2004a,b) and Saunders *et al.* (2006) present demonstration learning approaches based on supervised learning methods. Both groups use the k-nearest neighbor (KNN) (Mitchell, 1997) algorithm to classify instances based on similarity to training examples, resulting in a policy mapping from sensory observations to actions.

Chernova & Veloso (2007) represent the policy as a set of Gaussian mixture models, where each model, with multiple Gaussian components, corresponds to a single action.

Statistical supervised learning techniques that rely on labeled training data are frequently used in demonstration-based learning to learn a policy given a fixed set of labeled data (Bentivegna *et al.*, 2004a; Saunders *et al.*, 2006).

In our work we focus on a policy learning approach, in which the robot experiences the demonstration through its sensors while under the control of an expert. The task is executed by a computer-based teacher, and the details of the execution, in the form of paired observations and actions, are passed on to the robot. The robot then generalizes from these demonstrations in order to effectively execute the task itself.

## 3.3 Neural Network Model

An artificial neural network (ANN) (Ng, 2011; Rojas, 1996) is organized in layers and each layer is composed of a bunch of "neuron" nodes. A neuron is a compu-

tational unit that can reads inputs, processes them and generates an output, see Figure 3.1 as an example.

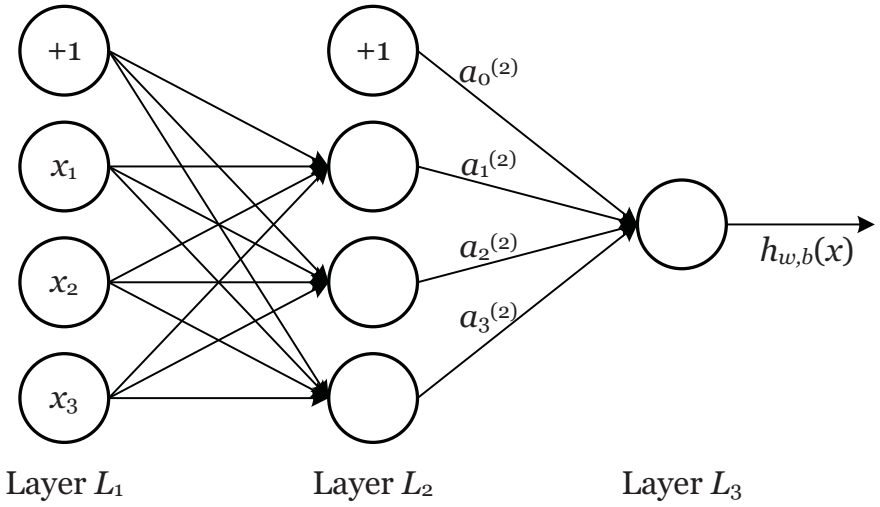



Figure 3.1: A neural network example.

The whole network is constructed by interconnecting many neurons. In this figure, one circle represents one neuron. The leftmost layer of the network is called the *input layer*, and the rightmost layer the *output layer*. The middle layer of nodes is called the *hidden layer*, since its values are not observed in the training set. The input and output layers serve respectively as the inputs and outputs of the neural network. The neurons labeled "+1" are called *bias units*. A bias unit has no input and always outputs +1. Hence, this neural network has 3 input units (excluding the bias unit), 3 hidden units (excluding the bias unit), and 1 output unit.

We use $n_l$ to denote the number of layers and label each layer $l$ as $L_l$. In Figure 3.1, $n_l = 3$, layer $L_1$ is the input layer, and $L_{n_l}$ is the output layer. The links connecting two neurons are named *weights*, representing the connection strength between the neurons. The parameters inside the neural network are $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$, where we write $W_{ij}^{(l)}$ to denote the weight associated with the connection between unit $j$ in layer $l$, and unit $i$ in layer $l+1$. Also, $b_i^{(l)}$ is the bias associated with unit $i$ in layer $l+1$. Thus, we have $W^{(1)} \in \mathbb{R}^{3\times 3}$ and $W^{(2)} \in \mathbb{R}^{1\times 3}$.*

*$b_i^{(l)}$ can also be interpreted as the connecting weight between the bias unit in layer $l$ who always outputs +1 and the neuron unit $i$ in layer $l+1$. Thus, $b_i^{(l)}$ may be replaced by $W_{i0}^{(l)}$. In

Each neuron in the network contains an *activation function* in order to control its output. We denote the activation of unit $i$ in layer $l$ by $a_i^{(l)}$. For the input layer $L_1$, $a_i^{(1)} = x_i$, the $i$-th input of the whole network. For the other layers, $a_i^{(l)} = f(z_i^{(l)})$. Here, $z_i^{(l)}$ denote the total weighted sum of inputs to unit $i$ in layer $l$, including the bias term (e.g., $z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)} x_j + b_i^{(1)}$), so that $a_i^{(l)} = f(z_i^{(l)})$.

Given a fixed setting of the parameters $(W, b)$, the neural network outputs a real number that is defined as the hypothesis $h_{W,b}(x)$. Specifically, the computation that this neural network represents is given by:

$$\begin{aligned} a_1^{(2)} &= f(W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + W_{13}^{(1)} x_3 + b_1^{(1)}), \\ a_2^{(2)} &= f(W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + W_{23}^{(1)} x_3 + b_2^{(1)}), \\ a_3^{(2)} &= f(W_{31}^{(1)} x_1 + W_{32}^{(1)} x_2 + W_{33}^{(1)} x_3 + b_3^{(1)}), \\ h_{W,b}(x) &= a_1^{(3)} = f(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)} + b_1^{(2)}). \end{aligned} \tag{3.1}$$

For a more compact expression, we can extend the activation function $f(\cdot)$ to apply to vectors in an element-wise fashion, i.e., $f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$, then we can write the equations above as:

$$\begin{aligned} a^{(1)} &= x, \\ z^{(2)} &= W^{(1)} a^{(1)} + b^{(1)}, \\ a^{(2)} &= f(z^{(2)}), \\ z^{(3)} &= W^{(2)} a^{(2)} + b^{(2)}, \\ h_{W,b}(x) &= a^{(3)} = f(z^{(3)}). \end{aligned} \tag{3.2}$$

$x = [x_1, x_2, x_3]^\top$ is a vector of values from the input layer. This computational process, from inputs to outputs, is called *forward propagation*. More generally, given any layer $l$'s activation $a^{(l)}$, we can compute the activation $a^{(l+1)}$ of the next layer $l + 1$ as:

$$\begin{aligned} z^{(l+1)} &= W^{(l)} a^{(l)} + b^{(l)}, \\ a^{(l+1)} &= f(z^{(l+1)}). \end{aligned} \tag{3.3}$$

this way, $W^{(1)} \in \mathbb{R}^{3\times 4}$ and $W^{(2)} \in \mathbb{R}^{1\times 4}$.

In this dissertation, we will choose $f(\cdot)$ to be the sigmoid function $f : \mathbb{R} \mapsto ]-1, +1[$ :

$$f(z) = \frac{1}{1 + \exp(-z)}. \tag{3.4}$$

Its derivative is given by

$$f'(z) = f(z)(1 - f(z)). \tag{3.5}$$

The advantage of putting all variables and parameters into matrices is that we can greatly speed up the calculation speed by using matrix-vector operations.

Neural networks can also have multiple hidden layers or multiple output units. Taking Figure 3.2 as an example, this network has two hidden layers $L_2$ and $L_3$ and two output units in layer $L_4$.

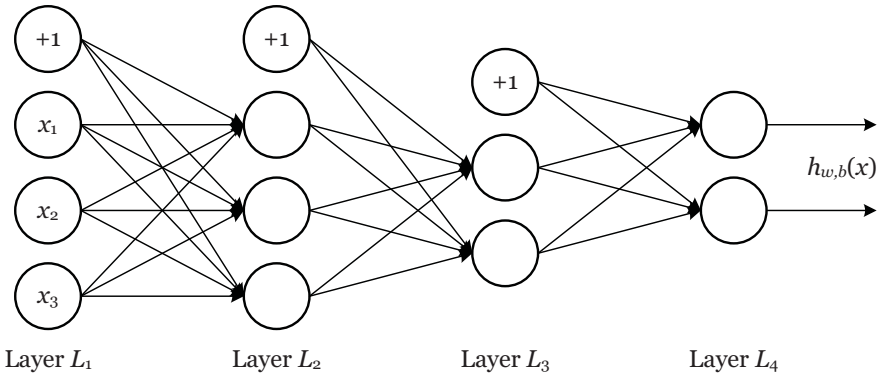



Figure 3.2: A neural network example with two hidden layers.

The forward propagation applies to all architectures of feedforward neural networks, i.e., to compute the output of the network, we can start with the input layer $L_1$, and successively compute all the activations in layer $L_2$, then layer $L_3$, and so on, up to the output layer $L_{n_l}$.

### 3.3.1 Backpropagation Algorithm

Suppose we have a fixed training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of $m$ training examples. We can train our neural network using batch gradient descent. In detail, for a single training example $(x, y)$, we define the cost function with respect to that single example to be:

$$J(W,b;x,y) = \frac{1}{2}\|h_{W,b}(x) - y\|^2. \tag{3.6}$$

This is a squared-error cost function. Given a training set of $m$ examples, we then define the overall cost function $J(W,b)$ to be:

$$\begin{aligned} J(W,b) &= \left[\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2 \\ &= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\|h_{W,b}(x^{(i)}) - y^{(i)}\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2. \end{aligned} \tag{3.7}$$

$s_l$ denotes the number of nodes in layer $l$ (not counting the bias unit). The first term in the definition of $J(W,b)$ is an average sum-of-squares error term. The second term is a regularization term that tends to decrease the magnitude of the weights, and helps prevent overfitting. Regularization is applied only to $W$ but not to $b$. $\lambda$ is the regularization parameter which controls the relative importance of the two terms. Note that $J(W,b;x,y)$ is the squared error cost with respect to a single example; while $J(W,b)$ is the overall cost function that includes the regularization term.

The goal of the backpropagation is to minimize $J(W,b)$ as a function of $W$ and $b$. To train the neural network, we first initialize each parameter $W_{ij}^{(l)}$ and each $b_i^{(l)}$ to a small random value near zero, and then apply an optimization algorithm such as batch gradient descent. It is important to initialize the parameters randomly, rather than to all 0's. If all the parameters start off at identical values, then all the hidden layer units will end up learning the same function of the input. More formally, $W_{ij}^{(1)}$ will be the same for all values of $i$, so that $a_1^{(2)} = a_2^{(2)} = a_3^{(2)} = \ldots$ for any input $x$. The random initialization serves the purpose of symmetry breaking.

One iteration of gradient descent updates the parameters $W, b$ as follows:

$$\begin{aligned} W_{ij}^{(l)} &= W_{ij}^{(l)} - \alpha\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b), \\ b_i^{(l)} &= b_i^{(l)} - \alpha\frac{\partial}{\partial b_i^{(l)}}J(W,b). \end{aligned} \tag{3.8}$$

The parameter $\alpha$ is the learning rate. It determines how fast $W$ and $b$ move towards their optimal values. If $\alpha$ is very large, they may miss the optimal and diverge. If $\alpha$ is tuned too small, the convergence may need a long time.

The key step in Equation (3.8) is computing the partial derivatives terms of the overall cost function $J(W,b)$. Derived from Equation (3.7), we can easily obtain:

$$\begin{aligned}\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b) &= \left[\frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b;x^{(i)},y^{(i)})\right]+\lambda W_{ij}^{(l)},\\ \frac{\partial}{\partial b_i^{(l)}}J(W,b) &= \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial b_i^{(l)}}J(W,b;x^{(i)},y^{(i)}).\end{aligned} \tag{3.9}$$

One of the main tasks of the backpropagation algorithm is to compute the partial derivatives terms $\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b;x^{(i)},y^{(i)})$ and $\frac{\partial}{\partial b_i^{(l)}}J(W,b;x^{(i)},y^{(i)})$ in Equation (3.9).

The backpropagation algorithm for one training example is shown as follows:

1. Perform a forward propagation, computing the activations for layers $L_2$, $L_3$, and so on up to the output layer $L_{n_l}$.

2. For each output unit $i$ in the output layer $n_l$, set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}}\left(\frac{1}{2}\|y-h_{W,b}(x)\|^2\right) = -(y_i-a_i^{(n_l)})\cdot f'(z_i^{(n_l)}). \tag{3.10}$$

3. For $l = n_l-1, n_l-2, n_l-3, \ldots, 2$:

   for each node $i$ in layer $l$, set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)}\delta_j^{(l+1)}\right) f'(z_i^{(l)}). \tag{3.11}$$

4. Compute the desired partial derivatives, which are given as:

$$\begin{aligned}\frac{\partial}{\partial W_{ij}^{(l)}}J(W,b;x,y) &= a_j^{(l)}\delta_i^{(l+1)},\\ \frac{\partial}{\partial b_i^{(l)}}J(W,b;x,y) &= \delta_i^{(l+1)}.\end{aligned} \tag{3.12}$$

Given a training example $(x, y)$, we first run a forward propagation to compute all the activations throughout the network, including the output value of the hypothesis $h_{W,b}(x)$. Then, for each node $i$ in layer $l$, we compute an error term $\delta_i^{(l)}$ that measures how much that node was "responsible" for any errors in our output. For an output node, we can directly measure the difference $\delta_i^{(n_l)}$ between the network's activation and the true target value, and for hidden units, we compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input.

In practice, we use matrix-vectorial operations to reduce the computational cost. We use "$\circ$" to denote the element-wise product operator [*]. By definition, if $C = A \circ B$, then

$$(C)_{ij} = (A \circ B)_{ij} = (A)_{ij} \cdot (B)_{ij}.$$

The algorithm for one can then be written:

1. Perform a forward propagation, computing the activations for layers $L_2$, $L_3$, up to the output layer $L_{n_l}$, using the equations defining the forward propagation steps.

2. For the output layer $n_l$, set

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \circ f'(z^{(n_l)}). \tag{3.13}$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, set

$$\delta^{(l)} = \left((W^{(l)})^\top \delta^{(l+1)}\right) \circ f'(z^{(l)}). \tag{3.14}$$

4. Compute the desired partial derivatives:

$$\begin{aligned} \nabla_{W^{(l)}} J(W, b; x, y) &= \delta^{(l+1)} \left(a^{(l)}\right)^\top, \\ \nabla_{b^{(l)}} J(W, b; x, y) &= \delta^{(l+1)}. \end{aligned} \tag{3.15}$$

In steps 2 and 3 above, we need to compute $f'(z_i^{(l)})$ for each value of $i$. Assuming $f(z)$ is the sigmoid activation function, we would already have $a_i^{(l)}$ stored

[*] Also called the Hadamard product.

away from the forward propagation throughout the whole network. Thus, using the Equation (3.5) for $f'(z)$, we can compute this as $f'(z_i^{(l)}) = a_i^{(l)}(1 - a_i^{(l)})$.

After getting all the partial derivatives that we desire, we can finally implement the gradient descent algorithm. One iteration of batch gradient descent is processed as follows:

1. Set $\Delta W^{(l)} := 0$, $\Delta b^{(l)} := 0$ (matrix/vector of zeros) for all $l$.

2. For $i = 1$ to $m$,

   (a) Use backpropagation to compute $\nabla_{W^{(l)}} J(W, b; x, y)$ and $\nabla_{b^{(l)}} J(W, b; x, y)$.

   (b) Set $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W, b; x, y)$.

   (c) Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W, b; x, y)$.

3. Update the parameters:

$$\begin{aligned} W^{(l)} &= W^{(l)} - \alpha \left[ \left( \frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right], \\ b^{(l)} &= b^{(l)} - \alpha \left[ \left( \frac{1}{m} \Delta b^{(l)} \right) \right]. \end{aligned} \tag{3.16}$$

$\Delta W^{(l)}$ is a matrix of the same dimension as $W^{(l)}$, and $\Delta b^{(l)}$ is a vector of the same dimension as $b^{(l)}$.

To train the neural network, we can repeatedly take steps of gradient descent to reduce our cost function $J(W, b)$.

## 3.4 Policy Learning from Demonstrations

Providing multiple expert demonstrations to mobile robots, we propose a policy learning method that largely ease the burden of manual programming. An artificial neural network is applied to generalize the undemonstrated states so as to give out an efficient policy for robot moving (Xia *et al.*, 2015).

### 3.4.1 Dataset Extraction from Demonstrations

The purpose of our learning from demonstration method is to derive a policy for the robot autonomous navigation tasks using dataset extracted by observing expert demonstrations.

Recall that a policy in an MDP is a mapping between states of the environment and robot actions. Thus, collected dataset should be composed of effective state-action pairs.

In our autonomous navigation tasks, all the sample data are extracted and collected from the expert demonstrations of optimal trajectories. We are given $M$ demonstration trajectories of length $T_i$, for $i = 1, ..., M$. Each trajectory is a sequence of state-action pairs, denoted

$$\mathcal{D}^{(i)} = \{(X_t^{(i)}, Y_t^{(i)})\}_{0 \le t \le T_i - 1}.$$

The robot can either observe the expert choices of state-action pairing, or try to repeat the expert trajectories by itself. Then the robot records the state and the corresponding action during each moving step.

The mobile robot is presented in Section 2.5. In order to bear the noise of sensor measurements, the reading distances from the sensors are not used directly as a state representation. Instead we introduce the *degree of danger**, defined in Table 3.1, to express the sensor readings, and each sensor reading corresponds to one degree of danger. The bigger the degree is, the more dangerous situation the robot is encountering, and a degree of 7 means a collision. Thus, the information of sensors at time instant $t$ are stored in a vector

$$D_t = [d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9]^\top,$$

where $d_i$ is the $i$-th sensor degree of danger.

Besides sensor readings, a state should include two more parameters. One is the target region, $V_t \in \{1, 2, ..., 9\}$, represented by one robot sensor region in Figure 2.7. The other one is an indicator $I_t \in \{0, 1\}$ that determines if the robot has detected the target. $I_t = 1$ means the target shows up in the robot sensors.

*According to different sensor types and detection ranges, the degree of danger can be defined alternatively.

| Degree of danger $d$ | Reading distance $r$ |
|---|---|
| 7 | $r < 3$ cm |
| 6 | $3 \leq r <7$ cm |
| 5 | $7 \leq r <11$ cm |
| 4 | $11 \leq r <15$ cm |
| 3 | $15 \leq r <20$ cm |
| 2 | $20 \leq r <25$ cm |
| 1 | $25 \leq r <30$ cm |
| 0 | $r \geq 30$ cm |

Table 3.1: Definition of the degree of danger

A complete state representation at time instant $t$ is defined as

$$X_t = \begin{bmatrix} D_t \\ V_t \\ I_t \end{bmatrix} \in \mathbb{R}^{11\times 1}. \tag{3.17}$$

The first 9 components are sensory degrees of danger, the $10^{\text{th}}$ is the target region, and the $11^{\text{th}}$ the indicator of the target detection.

The robot movement is discretized into five actions: moving forward (F), turning left at 30° (L30), turning left at 60° (L60), turning right at 30° (R30), and turning right at 60° (R60). For each state $X_t$, the robot action undertaken in the demonstrations is defined as $u_t \in \{1, 2, ..., 5\}$ that specifies the action among F ($u_t = 1$), L30 ($u_t = 2$), L60 ($u_t = 3$), R30 ($u_t = 4$) and R60 ($u_t = 5$).

In the dataset extracted from the demonstrations, in order to represent the robot action, we construct a label vector $Y_t = [y_1, y_2, y_3, y_4, y_5]^\top$ where

$$y_k = \begin{cases} 1, & \text{if } k = u_t \\ 0, & \text{elsewhere} \end{cases} \qquad \text{for } k = 1, ..., 5.$$

For example, an action 'R30' is labeled by $Y = [0, 0, 0, 1, 0]^\top$.

After all the demonstrations, the robot have collected a complete raw dataset of state-action pairs $\{\mathcal{D}^{(i)}\}_{1\leq i\leq M}$. However, not all of them are considered as *effective* data. Only a world state where at least one robot sensor has detected obstacles and the related robot action are considered as an effective state-action pair, i.e., a state $X_t$ with its components $D_t = [0, 0, 0, 0, 0, 0, 0, 0, 0]$, called a safe

state, and together with its corresponding action are considered as a "useless" pair and is excluded from the effective dataset. Therefore, we collect $M$ demonstration trajectories with an effective length* $N_i$, for $i = 1, ..., M$. We give the notation $\mathcal{O}^{(i)} = \{(X_t^{(i)}, Y_t^{(i)})\}_{0 \le t \le N_i - 1}$ to the effective dataset.

### 3.4.2 Architecture of Neural Network

In our method of policy learning from multiple demonstrations, a three-layer neural network is applied as a supervised learning tool to train effective dataset in the demonstrations. The architecture is shown in Figure 3.3.

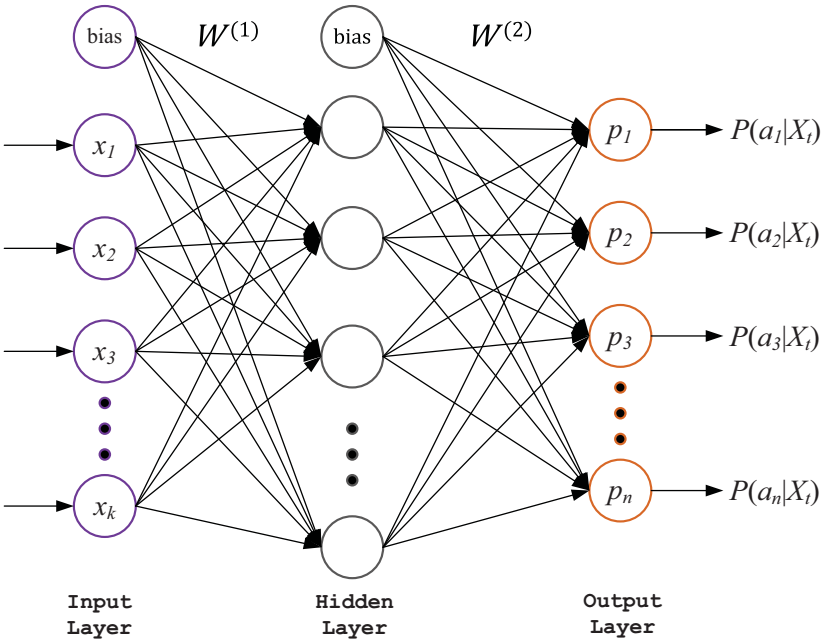



Figure 3.3: The three-layer neural network architecture in policy learning from demonstrations.

It has 11 neurons in the input layer and 5 neurons in the output layer. For a training example $(X_t^{(i)}, Y_t^{(i)})$, the state $X_t^{(i)} = \{x_j\}_{1 \le j \le 11}$ is sent to the input layer, and the network outputs a vector of 5 action probabilities $P(a_n \mid X_t) = \{p_n\}_{1 \le n \le 5}$, where $\{a_n\}_{1 \le n \le 5}$ corresponds with the actions {F, L30, L60, R30, R60}. $Y_t^{(i)}$ serves as the output label for the neural network training. Moreover, 8 neurons are designed in the hidden layer.

*Here, the expression "effective length" means the length of effective state-action pairs in one demonstration trajectory.

From now on, we consider the bias units as parts of the input and hidden layers, and they are set to 1. Hence, the input is now $\{x_j\}_{0\leq j\leq 11}$ where $x_0 = 1$ is the bias unit, and the same goes for the hidden layer.

The weight $W^{(1)}$ is used to connect the input layer and the hidden layer, and similarly, the weight $W^{(2)}$ links the hidden layer and the output layer.

$$W^{(1)} = \begin{bmatrix} \omega_{1,0}^{(1)} & \dots & \omega_{1,11}^{(1)} \\ \vdots & \ddots & \vdots \\ \omega_{8,0}^{(1)} & \dots & \omega_{8,11}^{(1)} \end{bmatrix} \in \mathbb{R}^{8\times 12}, \quad W^{(2)} = \begin{bmatrix} \omega_{1,0}^{(2)} & \dots & \omega_{1,8}^{(2)} \\ \vdots & \ddots & \vdots \\ \omega_{5,0}^{(2)} & \dots & \omega_{5,8}^{(2)} \end{bmatrix} \in \mathbb{R}^{5\times 9}. \quad (3.18)$$

$\omega_{i,j}^{(1)}$ represents the weight between $j$-th input neuron and $i$-th hidden neuron, and specifically, the bias unit in the input layer is expressed with $j = 0$. Similarly, $\omega_{i,j}^{(2)}$ represents the weight between $j$-th hidden neuron and $i$-th output neuron, and the bias unit in the hidden layer is also expressed with $j = 0$. Taking the bias units into account, $W^{(1)}$ has a size of $8 \times 12$, and respectively, $W^{(2)}$ has a size of $5 \times 9$.

For the hidden and output units, we use the sigmoid function as the activation function, see Equation (3.4).

### 3.4.3 Policy Learning Process

A neural network training process can generalize over the dataset of available training examples such that valid solutions are also acquired for similar states that may not have been encountered during demonstrations.

#### 3.4.3.1 Neural Network Training

The feedforward neural network (FFNN) takes the responsibility to calculate the predicted probabilities of actions from an state input $X_t = \{x_i\}_{0\leq i\leq 11}$. The weights $W^{(1)}$ and $W^{(2)}$ in Equation (3.18) keep unchanged. The hidden units are calculated as follows:

$$z_j^{(2)} = \sum_{j,i} \omega_{j,i}^{(1)} \times x_i, \quad \text{for } i = 0, \ldots, 11,$$
$$h_j = \frac{1}{1 + e^{-z_j^{(2)}}}. \tag{3.19}$$

where $x_0 = 1$ is the bias unit, and $h_j \in [0, 1], j = 1, \ldots, 8$ are the hidden units.

Then, adding the bias unit $h_0 = 1$ into the hidden layer, the output layer is calculated as follows:

$$z_k^{(3)} = \sum_{k,j} \omega_{k,j}^{(2)} \times h_j, \quad j = 0, \ldots, 8, \tag{3.20}$$

$$p_k = \frac{1}{1 + e^{-z_k^{(3)}}}. \tag{3.21}$$

where $p_k \in [0, 1], k = 1, \ldots, 5$ are the output units.

In future autonomous navigation tasks, FFNN is also applied in each state to calculate which action to take by selecting the biggest value of probabilities $p_i$.

FFNN outputs the predicted action probabilities and these need to be compared with desired output label $Y_t = \{y_k\}_{1 \leq k \leq 5}$.

The backpropagation neural network (BPNN) is designed to train the neural network by updating the weights $W^{(1)}$ and $W^{(2)}$. Only a neural network with fully trained weights can be used in robot navigation tasks. The weight changes are based on the network's error, the difference between its output for a given input and a target value, what the network is expected to output. The output label in LfD dataset is treated as the target for the output unit corresponding to the selected action. We use the cross-entropy error to define the cost function $J(W)$:

$$J(W) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{5} y_k^{(i)} \cdot \log p_k^{(i)} + \left(1 - y_k^{(i)}\right) \cdot \log\left(1 - p_k^{(i)}\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\omega_{j,i}^{(l)}\right)^2 \tag{3.22}$$

where $m$ is the total number of training examples (effective state-action pairs). $y_k$ is the output label extracted from demonstrations, and $p_k$ is the action probabilities via FFNN. $L$ is the total number of layers in network, and $s_l$ is the number of units in layer $l$, excluding bias units. The second term of $J(W)$ is the regularization term aimed at avoiding overfitting or underfitting, and $\lambda$ is regularization parameter.

The task of BPNN is to find min $J(W)$. Based on the gradient descent method presented in Section 3.3.1, the network error is optimized so as to update the weight. We first calculate the errors of all neurons in each training example:

$$\delta_k^{(3)} = p_k - y_k, \qquad k = 1, \ldots, 5 \tag{3.23}$$

$$\delta_j^{(2)} = \left((W^{(2)})^T \cdot \delta^{(3)}\right)_j \cdot h_j(1 - h_j), \quad j = 0, \ldots, 8 \tag{3.24}$$

$\delta_j^{(2)}$ is the error of $j$-th neuron in the hidden layer. $\delta_k^{(3)}$ is the error of $k$-th neuron in the output layer.

Then, for all the training examples, we calculate the errors of weights $\Delta_{i,j}^{(1)}$ and $\Delta_{i,j}^{(2)}$ using the increments:

$$\Delta_{k,j}^{(2)} := \Delta_{k,j}^{(2)} + h_j \delta_k^{(3)} \tag{3.25}$$

$$\Delta_{j,i}^{(1)} := \Delta_{j,i}^{(1)} + x_i \delta_j^{(2)} \tag{3.26}$$

Finally, we can obtain:

$$\frac{\partial J}{\partial \omega_{i,j}^{(l)}} = \begin{cases} \frac{1}{m}\Delta_{i,j}^{(l)} + \lambda \omega_{i,j}^{(l)}, & \text{if } j \neq 0 \\ \frac{1}{m}\Delta_{i,j}^{(l)}, & \text{if } j = 0 \end{cases} \tag{3.27}$$

We repeat the following step until the weights are updated to their optimal values:

$$W^{(l)} \leftarrow W^{(l)} - \alpha \frac{\partial J}{\partial W^{(l)}}. \tag{3.28}$$

We now put together the effective data that are extracted from all demonstrations and we establish a mapping $\pi : S \rightarrow A$ between state input $S$ and action output $A$.

$$S = \begin{bmatrix} X^{(1)} \\ \vdots \\ X^{(M)} \end{bmatrix}, \qquad A = \begin{bmatrix} Y^{(1)} \\ \vdots \\ Y^{(M)} \end{bmatrix}. \tag{3.29}$$

$M$ is the total number of demonstrations. Both $X^{(n)}$ and $Y^{(n)}$ are matrices with the following expressions:

$$X^{(n)} = \begin{bmatrix} x_{1,1}^{(n)} & \cdots & x_{1,11}^{(n)} \\ \vdots & \ddots & \vdots \\ x_{j,1}^{(n)} & \cdots & x_{j,11}^{(n)} \end{bmatrix}, \quad Y^{(n)} = \begin{bmatrix} y_{1,1}^{(n)} & \cdots & y_{1,5}^{(n)} \\ \vdots & \ddots & \vdots \\ y_{j,1}^{(n)} & \cdots & y_{j,5}^{(n)} \end{bmatrix}. \tag{3.30}$$

$s_{j,i}^{(n)}$ represents the $i$-th state component of $j$-th state-action pair extracted in $n$-th demonstration. $y_{j,k}^{(n)} \in \{0, 1\}$ expresses the $k$-th action label of $j$-th state-action pair extracted in $n$-th demonstration. Note that $j$ may vary in different demonstrations, since the number of effective pairs varies in different demonstrations.

The input $S$ and action label $A$ are sent to neural network. An advanced optimization method *fminunc* is used to optimize the network error and to update the weights. The neural network training eventually produces the well-learned weights $W^{(1)}$ and $W^{(2)}$, and allows the network to work in forward pass only for future robot navigation tasks in unknown environments.

### 3.4.4 Algorithm

We now present the algorithm of policy learning from multiple demonstrations in Algorithm 3.

This algorithm can also be applied to other robot learning problems.

## 3.5 Demonstration by Modified A* Algorithm

A* is a well known best-first shortest path finding algorithm, and is highly used in robotics. It is used to find a path from a given start node to a target node. It searches the whole map area and finds each possible path from the start node to reach the target node and then gives the shortest path. Unlike most A* algorithms use rectangle grids for the world representation, we propose a modified A* that

**Algorithm 3** Policy Learning from Multiple Demonstrations

**Input:** A set of demonstrations $\mathcal{D}$;
1: Initialize a 3-layer neural network with randomly generated weights;
2: **for** each demonstration $\mathcal{D}^{(i)}$ **do**
3: The robot observe or repeat the demonstration and collect a dataset of state-action pairs $\mathcal{D}^{(i)} = \{(s_t^{(i)}, a_t^{(i)})\}$;
4: **end for**
5: Gather and scale all the dataset;
6: Extract the effective data to create a training dataset $\mathcal{O}^{(i)} = \{(x_j^{(i)}, y_j^{(i)})\}$;
7: Train a neural network with the data;
8: Deploy and test the network on the robot;
**Output:** An optimal policy $\pi^*$.

is based on robot actions. Since a robot can choose 5 possible moving actions, one node in modified A* have 5 potential neighbor nodes according to actions. Without restriction of grid representation, the planning becomes more practical.

### 3.5.1 Node Representation

Each node is represented by a structure array with the following fields: (1) current position; (2) robot heading; (3) $H$ value; (4) $G$ value; (5) $F$ value; (6) parent position; (7) robot action.

Take *start node* as example: current position is the robot starting coordinates; heading is the robot initial heading; $H$ value is the distance to the target; $G$ value is 0; $F$ value is the sum of $H$ and $G$ values; it has no parent position and no action to take.

Only current position is recorded into the open list and the closed list. Other field values can be called on by searching the node with the corresponding current position.

Potential neighbor nodes are defined based on the robot actions.

Our work uses the classic representation of the A* algorithm as the evaluation function $F(x)$:

$$F(x) = G(x) + H(x). \tag{3.31}$$

$G(x)$ is the path cost function which calculates the actual total cost of the path to reach the current node from the start node. $G$ cost is accumulated by

the total distance traveled by all the past moving steps.

$H(x)$ is the heuristic function which estimates the cost of the path from the current node to the goal node. $H$ cost is simply taken as the direct distance from the current position to the target regardless of obstacles.

The node representation is updated according to the following rule: define parent position of the neighbor as current position of current node; define $G$ value of the neighbor as tentative $G$ value; define $H$ value of the neighbor as distance to target; define $F$ value of the neighbor as sum of $G$ and $H$; define heading of the neighbor as the heading after robot movement; define action of the neighbor as the action the robot take to move from current to this neighbor.

### 3.5.2 Algorithm

The core the modified A* algorithm is a heuristic search based on robot actions, not on world grids. A node within the reach of one possible action is considered as a potential neighbor node of the current node when the nine robot sensors detect no collision possibility of obstacles in that node. Then the potential neighbor nodes will be evaluated to determine whether move it to open list or not. The pseudo code is shown in Algorithm 4.

### 3.5.3 Result

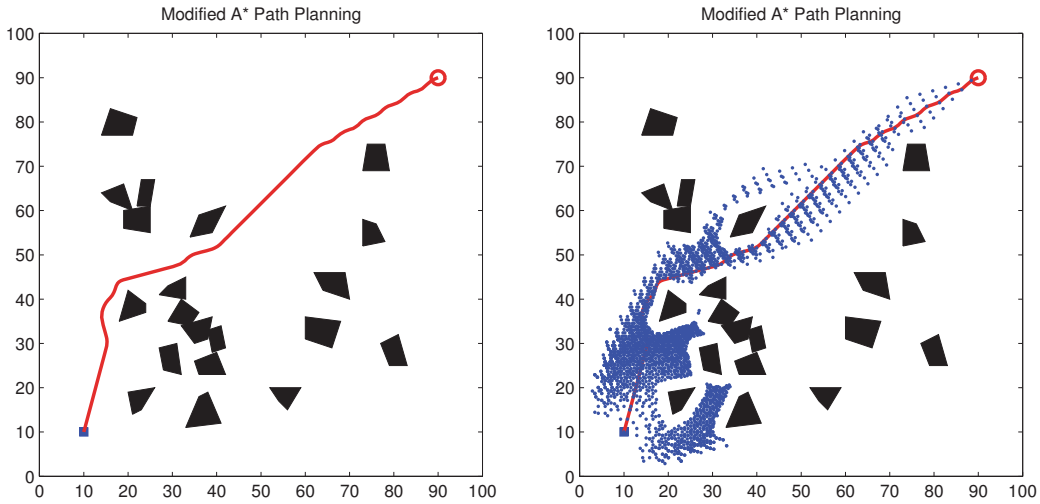
The advantage of this modified A* algorithm is that the path finding is no longer restricted to a grid world and this method can be adapted to any real robot applications with any robot action spaces.

In Figure 3.4, the start point and the target are respectively placed at $(10, 10)$ (a solid blue rectangle) and $(90, 90)$ (a hollow red circle). The optimal path is planned by the modified A* algorithm, and represented by red stars in Figure 3.4(a). The blue dots alongside the red path in Figure 3.4(b) showed us all the world states visited by our algorithm. Without dividing the environment into grids, the proposed algorithm has successfully programmed an optimal path.

In Figure 3.5, we compare our method (in red solid curves) and the conventional A* algorithm (in green dashed curves). For the same navigation environment, the conventional A* used 48.1936 s to plan a path of 124.3638 m, while our algorithm took 23.2288 s to give out a path of 120.2053 m. We could tell that

**Algorithm 4** Modified A* algorithm

```
1: Initialize closed list and open list as empty sets.
2: Create a first node structure start node, and add it to the open list.
3: repeat
4:     Define current node as the node in open list having the lowest F value.
5:     Add current node to closed list, and then remove it from open list.
6:     if current node is close enough to the target then
7:         Break.
8:     end if
9:     Evaluate the list of potential neighbor nodes according to their probabil-
   ities of collision.
10:    if exist collision probability then
11:        Remove this node from the list of neighbor nodes.
12:    else
13:        Keep this node in the list of neighbor nodes.
14:    end if
15:    for each neighbor node do
16:        if neighbor node is in closed list then
17:            Continue.
18:        end if
19:        Define tentative G value as the sum of the G value of current node
   and the distance between current node and this neighbor node.
20:        if neighbor is not in open list or tentative G value < G value of this
   neighbor then
21:            Update the node representation.
22:            if this neighbor node is not in open list then
23:                Add the neighbor to open list.
24:            end if
25:        end if
26:    end for
27: until open list is empty
28: Create an empty set of structure Path.
29: Add current node to Path.
30: repeat
31:    Define Next as the parent position of current node.
32:    Find the node in the closed list whose current position is Next.
33:    Update current node as this node.
34:    Add current node to Path.
35: until current node is start node
```




(a) Modifed A* planning

(b) Modifed A* with all nodes in the closed list (blue dots)

Figure 3.4: Modified A* Algorithm Result.

this modified method produced a shorter and more optimal path, and specifically saved much planning time. In sum, our method is more practical.

## 3.6 Experimental Results

Given partial knowledge about its environment and a goal position or series of positions, navigation encompasses the ability of the robot to act based on its knowledge and sensor values so as to reach its goal positions as efficiently and as reliably as possible.

Simulation experiments are carried out in order to evaluate the proposed method. The mobile robot is represented by a blue rectangle-shaped object. It is equipped with 9 sensors to observe the surrounding environment, as shown in Figure 2.6. The environment map has a size of 100 m $\times$ 100 m. The obstacles are randomly scattered in the environment. The initial position of the robot is placed at $(10, 10)$ and the target position, a red circle in the map, is found at $(90, 90)$. The velocity of the robot is fixed at 2 m/s. The mission of the robot is to start from the initial position and to find an optimal path to arrive at the target position without any collision with any obstacles. If no obstacles are detected, the robot is designed to move in an rough direction to the target.

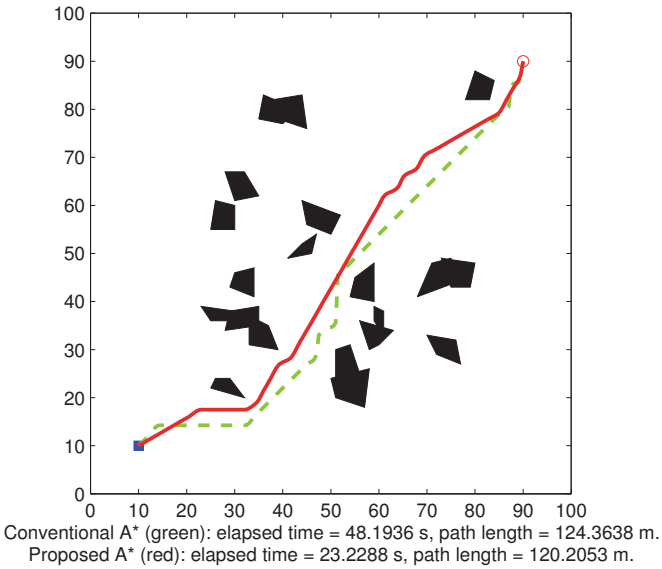



Figure 3.5: Pathfinding results using modified A* (red solid) and conventional A* (green dashed) algorithms.

The neural network has three layers: 11 in the input layer, 8 in the hidden layer and 5 in the output. The input is the state of the environment. The range of sensor detection is 10 m. The output is five action probabilities, and the robot will take the action with the biggest probability. Some important experiment parameters are selected as follows:

- Regularization parameter in Equation (3.22): $\lambda = 1$,
- Total number of demonstrations: $M = 25$,
- Total iterations of BPNN: $iter = 300$.

### 3.6.1 Policy Learning Process

In our work, 25 demonstrations are generated by modified A* algorithm. All of them have different configurations of random obstacle positions. The advantage of different maps is that the robot can always face new challenge and help gather more state-action combination. Some of the demonstrations are presented in Figure 3.6.

It is observed that the demonstrations can always provide optimal paths, and this will greatly improve the robot learning performance compared to human

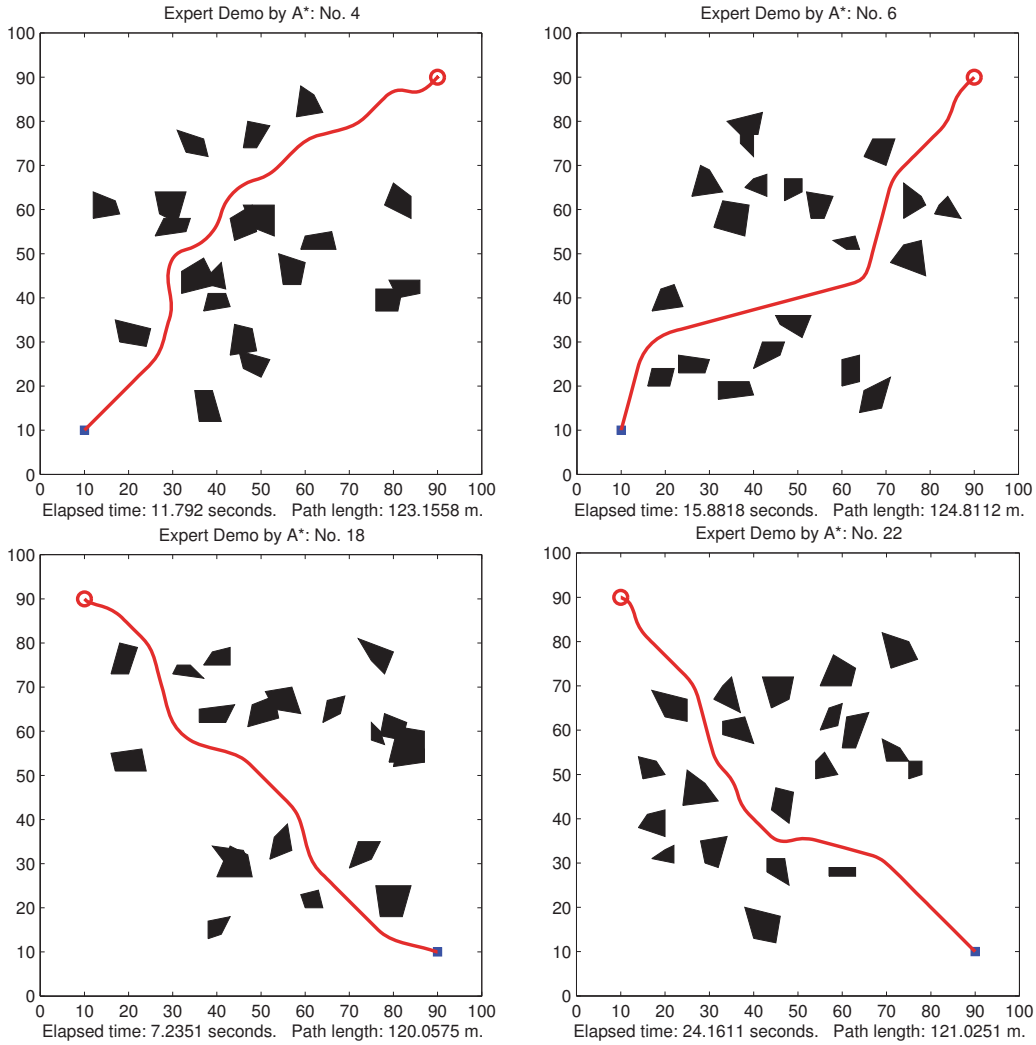



Figure 3.6: Expert demonstrations given by modified A* algorithm.

teachers who cannot guarantee the quality of their demonstrations. Modified A* algorithm has proven its success in global navigation tasks. However, in real world, a robot cannot always have a clear prior knowledge about the environment. That is why we conduct learning from demonstration process, hoping to transplant the ability of obstacle avoidance to future navigation tasks where only local environment information is available to robots. Neural network is applied as a supervised learning tool to train the data collected from the demonstration. The goal of the neural network training is to minimize the cost $J(W)$ in Equation (3.22) by iteratively updating the weights $W^{(1)}$ and $W^{(2)}$. The result is shown in

Figure 3.7.

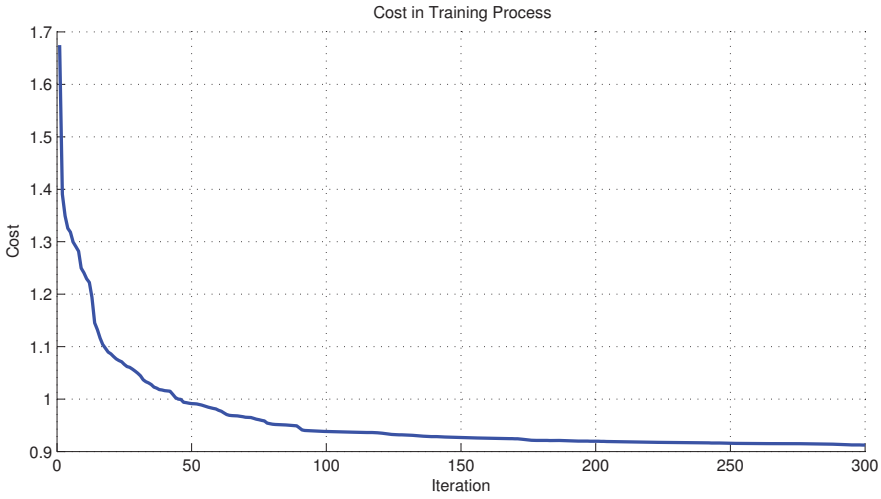



Figure 3.7: Changes of cost $J(W)$ in learning process.

In the first 50 iterations, the cost decreases fast and then becomes smaller stably. Finally after all 300 iterations, the cost converges and stays around 0.91. Once the training terminates, $W^{(1)}$ and $W^{(2)}$ stop updating and the learning process from demonstration is accomplished. It is now the time to examine the robot learning achievement.

### 3.6.2 Robot navigation in unknown environments

The robot is now placed in a new and unknown world to execute a navigation task. Unlike the demonstrations, the robot has no prior knowledge of the numbers, sizes and positions of the obstacles. It does not know either the exact target position but only a rough direction. It has a maximum of 500 moving steps and it chooses the best fit action in each step according to action probabilities calculated via FFNN. Figure 3.8 shows four different navigation tasks. Although the environments varied, the robot succeeded in getting to the target in a collision free path, and always kept a safe distance to surrounding obstacles.

It is also observed that the paths that the robot chose may not be the optimal ones due to a lack of complete knowledge of environment map, but they are still fully acceptable and perfect enough to meet our expectation. Therefore, the above experiments have proven the stability of the proposed learning from demonstration method.

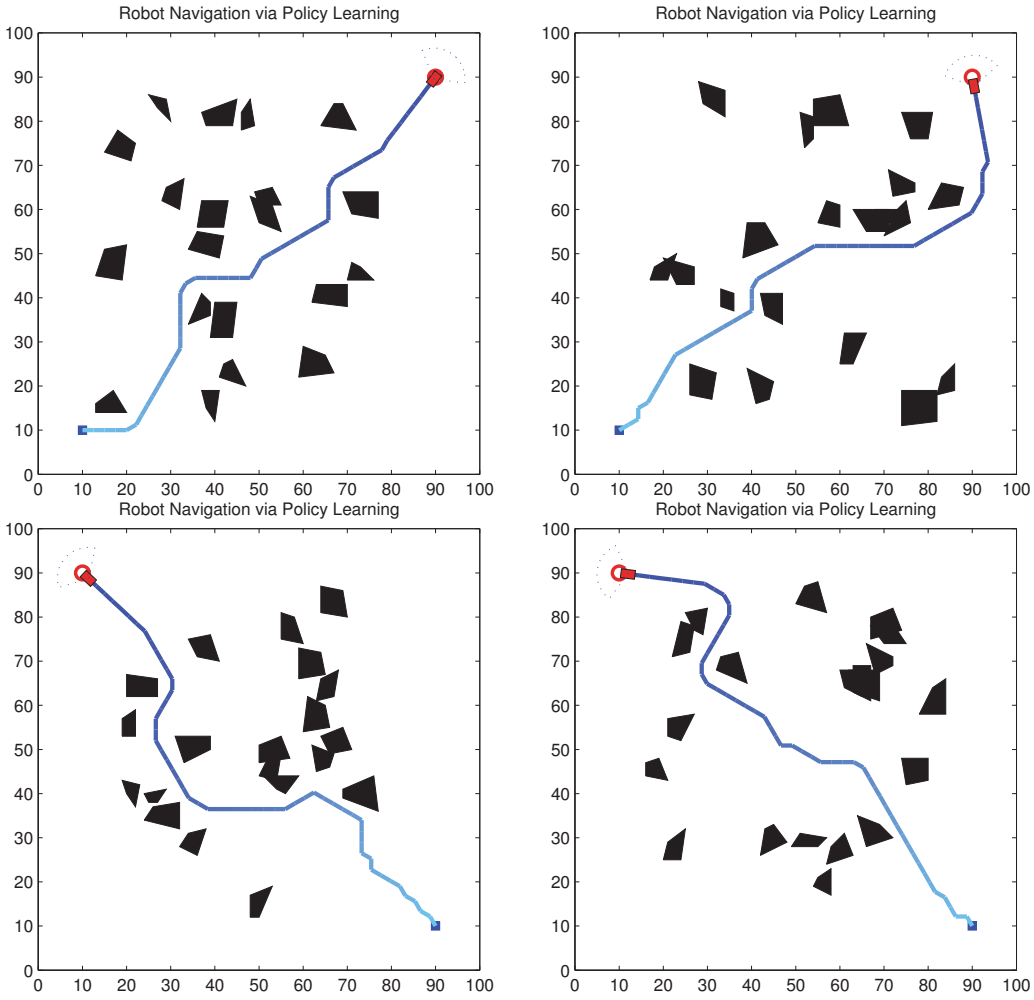



Figure 3.8: Robot navigation in new environments.

### 3.6.3 Paths Comparison

Since the paths are not the optimal ones, we are now investigating the difference between them. Given an unknown world, we first plan an optimal path in red curve using our modified A* algorithm in a global view, and then let the robot navigate by the learned policy. Figure 3.9 tells that the robot has learned to find a quasi-optimal path unless sometimes it thinks dangerous to pass through a narrow path between obstacles, such as in the last navigation map. Also, after training the robot, the robot can save much time in real-time navigation.

The robot may not always choose a similar path as the expert suggests, as the last two figures in Figure 3.9. However, the robot could still navigate to the target without any collisions.

### 3.6.4 Autonomous Navigation in Dynamic Environments

We tested our policy learning algorithm in dynamic environments. The results are shown in Figure 3.10. In the environment, the black solid objects are static obstacles, and the purple rectangle ones are moving obstacles, which move in an unpredictable direction.

The robot first encountered a static obstacle at time 10, after avoiding it, the robot encountered several moving obstacles around him at time 23. The robot could successfully avoid them. Before arriving at the destination, the robot met another moving obstacle at time 47, and he kept a safe distance with it. Finally the robot reached the destination without any collisions. Therefore, the robot has successfully learned an control policy from the expert demonstration in autonomous navigation in dynamic environments.

### 3.6.5 Discussions

We compare the results if providing different number of demonstrations.

| Number of demos | 20 | 25 | 30 | 35 | 40 |
|---|---|---|---|---|---|
| Rate of success | 89% | 92% | 92% | 93% | 94% |

Table 3.2: Compassion of number of demonstration versus rate of success.

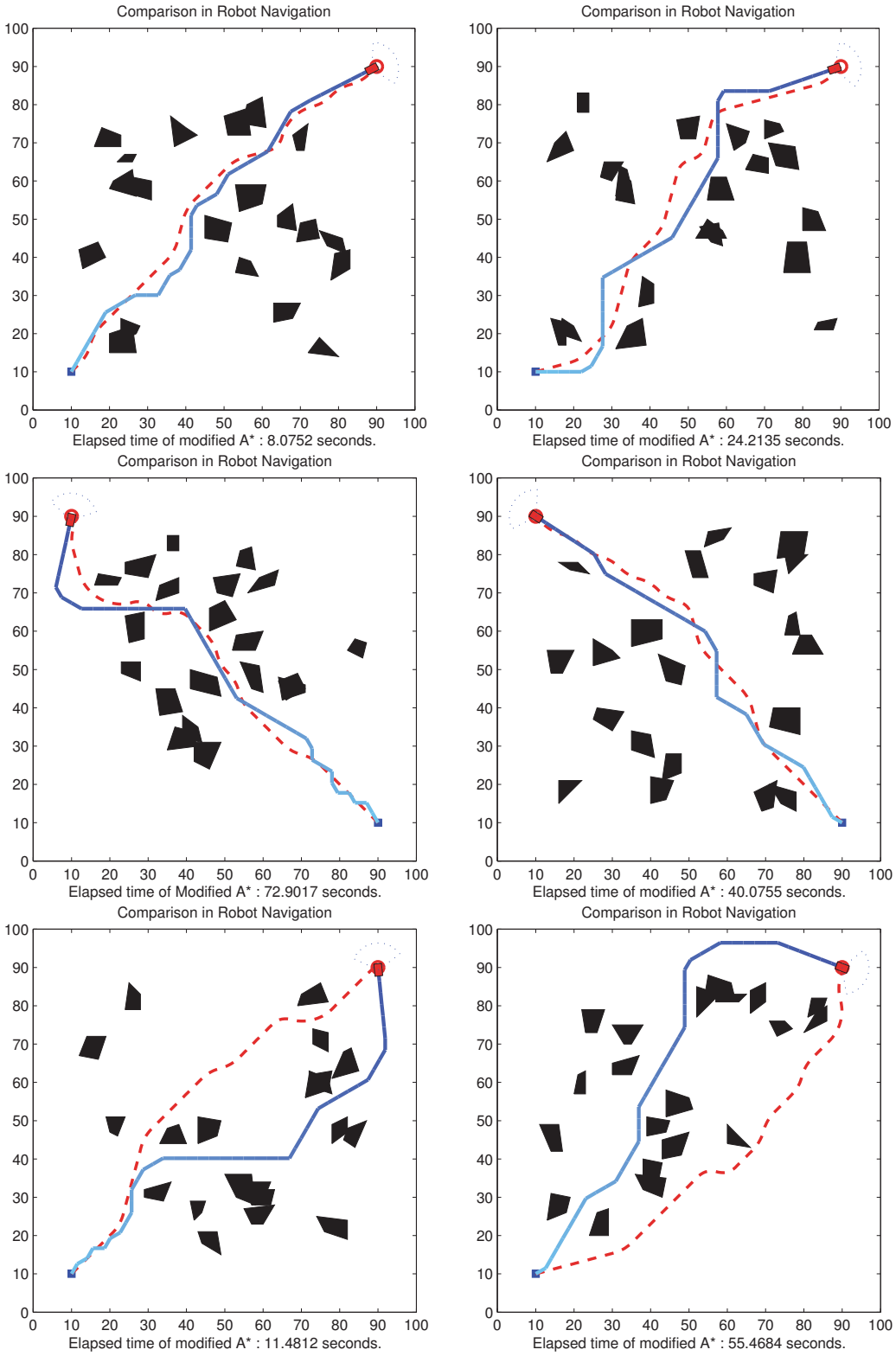



Figure 3.9: Path comparison in two methods. The red dotted curves are the suggested paths by computer expert, and The blue solid curves are actual robot navigation trajectories.

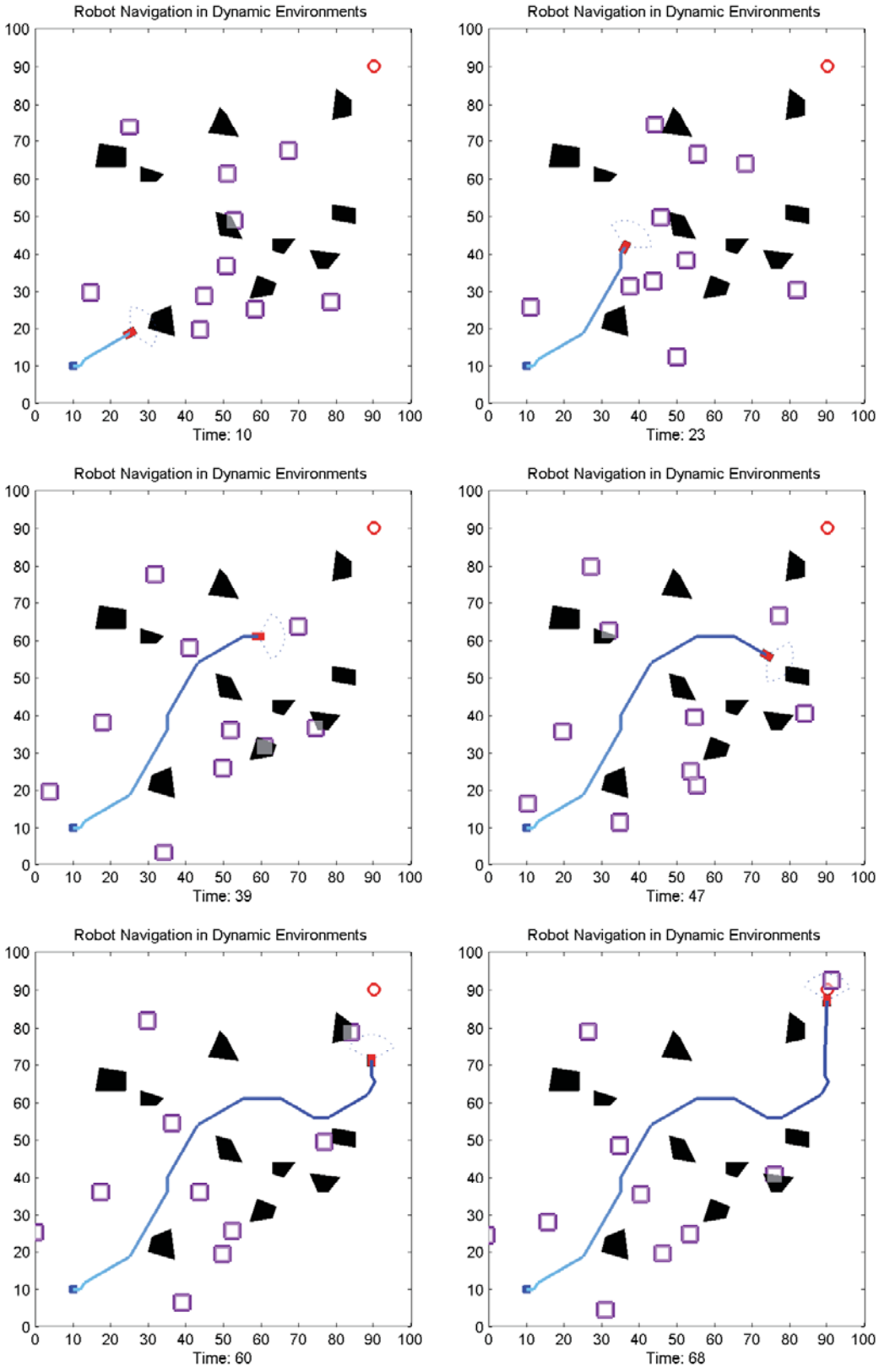



Figure 3.10: Autonomous robot navigation in a dynamic environment using proposed policy learning method.

We can see the rate of success change not that big, therefore, we can safely choose a number of demonstrations of 25.

We also give the results how the different numbers of iterations in backpropagation would influence the training performance.

| Number of iterations | 100 | 200 | 250 | 300 | 350 | 400 | 500 |
|---|---|---|---|---|---|---|---|
| Prediction accuracy(%) | 81.12 | 82.68 | 82.83 | 83.62 | 83.31 | 83.15 | 83.46 |
| Training time (s) | 3.42 | 6.82 | 8.75 | 10.64 | 12.77 | 13.64 | 17.08 |

Table 3.3: Compassion of number of iterations in training performance.

Using the same training set, we can see that from 250 iterations, the prediction accuracy stayed stable while the training time increased, and also more iterations would risk overfitting. Therefore, a number of iterations between 250 and 350 was a good choice.

From the above experiments, the robot successfully reproduces the task after learning a policy even when all objects are in novel positions. The proposed method using learning from demonstration has equipped a mobile robot with intelligence and adaptive abilities in performing an independent navigation task.

## 3.7 Conclusion

Building robust and reliable autonomous navigation systems that generalize across environments and operating scenarios remains a core challenge in robotics. Our motivation is for robot capabilities to be more easily extended and adapted to novel situations, even by users without programming ability.

Learning from demonstration is an approach aimed at recovering the instructions directly from teacher's demonstrations.

In our work, we proposed a method for mobile robots to learning an optimal policy offline from expert demonstrations, especially in the cases where human demonstrations are not available to obtain, and then to navigate online new environments autonomously. The neural network is applied to generalize the undemonstrated states. The experiments showed a good learning efficiency and generalization capability without lots of examples, and can eventually enable a mobile robot to learn a policy to autonomously execute future navigation tasks.

# Chapter 4

# Reinforcement Learning under Stochastic Policies

## Contents

## 4.1 Introduction

Endowing robots with human-like abilities to perform specific skills in a smooth and natural way is one of the important goals of robotics. In the previous chapter, we investigated that a mobile robot learned a control policy via multiple expert demonstrations. These demonstrations provided the robot with prior experience and our learning algorithm generalized to the undemonstrated states. However, in some cases, such as exploring hazardous environments, no prior experience or demonstrations are available to the mobile robots, and we still want them to learn how to behave in unknown environments. This is the main problem that we are dealing with in this chapter.

Reinforcement learning (RL) is the key tool that helps us to create robots that can learn new skills by themselves, just similarly to our human beings. Reinforcement learning is realized by interacting with an environment. In RL, the learner is a decision-making agent that takes actions in an environment and receives an reinforcement signal for its actions in trying to accomplish a task. The signal, well known as reward (or penalty), evaluates an action's outcome, and the agent seeks to learn to select a sequence of actions, i.e. a policy, that maximize the total accumulated reward over time.

Reinforcement learning can be formulated as a Markov Decision Process. Model-based RL algorithms can be used if we know the state transition function $T(s, a, s')$. In contrast, we focus more on model-free RL that the environment model $T$ is not known in advance, especially in our work of autonomous navigation tasks.

The whole learning scenario is a process of trial-and-error runs. We apply a Boltzmann probability distribution to tackle the problem of the exploration-exploitation tradeoff, that is, the dilemma between should we exploit the past experiences and select the actions that as far as we know are beneficial, or should we explore some new and potentially more rewarding states. Under the circumstances, the policies are stochastic.

Classic RL algorithms are usually applied to small sets of states and actions, that is, the agent can only visit a subset of the of states during the trial-and-error runs. In real applications, the state spaces are of a large scale and this will bring the problems of the generalization and the curse of dimensionality. We use the neural network structure to generalize and approximate the value of all the states. In the previous chapter, we have successfully implemented the neural network to train offline a control policy for mobile robots after all the state-action pairs are collected. In this chapter, we use the network to do incremental online learning without acquiring the output labels explicitly.

In this chapter, we present a RL based self-learning algorithm in the cases where expert demonstrations cannot provided to learning agents in advance and they need to find a robust policy via interacting with the environment (Xia & El Kamel, 2014a,c, 2015d). Experiments are conducted on the autonomous navigation tasks for mobile robots.

## 4.2 Related Work

Many researches on robot autonomous navigation using reinforcement learning were conducted. In general, reinforcement learning is realized by the interaction between the robot and the surrounding environment. Instead of explicitly detailing the solution to a problem, in RL, the designer provides feedback that measures the one-step performance of the robotic system for future self-improvement.

Yang *et al.* (2004) used continuous Q-learning algorithm for solving the autonomous navigation problem for mobile robots. They used a multilayer feedforward neural network to approximate the Q-learning value function. This method designed only three actions and was applied for static environments. Also, the robot was trained in the same environment, and no proof of the portability of the algorithm in other environments was provided.

Lagoudakis & Parr (2003) proposed the least-squares policy iteration (LSPI) that is motivated by the least-squares temporal-difference learning algorithm (LSTD) for prediction problems. This method can be efficiently used on batch reinforcement learning problems. Also for batch learning, neural fitted-Q iteration (NFQ) (Riedmiller, 2005) is a model-free, neural network based reinforcement learning algorithm to train Q-value functions. The main drawback of batch RL is

that it relys on the efficiency of a complete data collection, which is quite costly, especially when the agent cannot achieve its goal in the learning scenarios.

Similar methods using reinforcement learning and neural network were presented in (Huang *et al.*, 2005; Li *et al.*, 2006; Qiao *et al.*, 2008). These papers have demonstrated the feasibility of their algorithms, but they share some common problems. First, these papers used exactly the same environment both to train the robot learning ability and to test the navigation skills. However, what is expected is that the robot learns the obstacle avoidance behaviors in some environments and can navigate independently in a completely new unknown environment. Therefore, the robustness of their methods need to be proved. Second, limited number of state and action spaces and simple design for reward function bring the question of functionality of navigation tasks in more complicated environments. Moreover, the algorithms in (Huang *et al.*, 2005; Li *et al.*, 2006) were tested in static environments and only one dynamic obstacle was considered in (Qiao *et al.*, 2008).

Knudson & Tumer (2011) evaluated reactive and learning navigation algorithms for exploration robots and showed a better performance using neuro-evolutionary algorithm compared to rule-based algorithm. At the same time, they indicated that when the environments became particularly complex, the neuro-evolutionary algorithm was not able to establish a good policy for navigation.

Reinforcement learning are widely used in various robotic systems. A humanoid robot navigation was described in (Navarro-Guerrero *et al.*, 2012) by using a supervised RL approach combined with Gaussian distributed state activation, and the robot successfully performed a backward docking movement used for autonomous recharging. El-Fakdi & Carreras (2013) proposed a two-step gradient-based reinforcement learning control system for solving the action selection problem of an autonomous underwater vehicle in a visual-based cable tracking task. A neural network reinforcement learning method was studied for visual control of a robot manipulator (Miljković *et al.*, 2013). A direct mapping from the image space to the actuator command was developed by using two different RL algorithms, Q-learning and SARSA, combined with neural networks. A database of representative learning samples was also employed in order to speed

up the convergence of the algorithms. This hybrid system could provide a high accuracy of a manipulator positioning in a situation of low resolution images.

Q-learning and SARSA algorithms are two major RL techniques, but they possess different characteristics. SARSA has a faster convergence characteristics, while Q-learning has a better final performance. However, SARSA is easily stuck in the local minimum and Q-learning needs longer time to learn. Wang et al. proposed a method of combining Q-learning and SARSA algorithm, called backward Q-learning, to deal with the dilemma between exploration and exploitation for action selection policy (Wang *et al.*, 2013). This method can enhance learning speed and improve the final performance. Moldovan and Abbeel also addressed a method of safe exploration in Markov decision process when in uncertain dynamic environments (Moldovan & Abbeel, 2012).

Many other RL-related algorithms can be found in (Wiering & van Otterlo, 2012). Xu *et al.* (2014) presents also a comprehensive survey on recent developments in reinforcement learning algorithms with function approximation.

## 4.3 Model-Free Reinforcement Learning Methods

In our work, We study reinforcement learning and robot control problems in which a mobile robot acts in a stochastic environment by sequentially choosing actions over a sequence of time steps, in order to maximize a cumulative reward. We model the problem as a Markov Decision Process: a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition dynamics distribution $P(s_{t+1} \mid s_t, a_t)$ satisfying the Markov property $P(s_{t+1} \mid s_1, a_1, ..., s_t, a_t) = P(s_{t+1} \mid s_t, a_t)$, for any trajectory $s_1, a_1, s_2, a_2, ..., s_T, a_T$ in state-action space, and a reward function $r : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$. A stochastic policy $\pi(s_t, a_t) = P(a_t \mid s_t)$ is used to select actions and produce a trajectory of states, actions and rewards $s_1, a_1, r_1, s_2, a_2, r_2, ..., s_T, a_T, r_T$ over $\mathcal{S} \times \mathcal{A} \times \mathbb{R}$.

An on-policy method learns the value of the policy that is used to make decisions. The value functions are updated using results from executing actions determined by some policy. An off-policy methods can learn the value of the optimal policy independently of the agent's actions. It updates the estimated value functions using hypothetical actions, those which have not actually been tried.

We focus on model-free RL methods that the robot drives an optimal policy without explicitly learning the model of the environment. Q-learning (Watkins, 1989) and Sarsa (Rummery & Niranjan, 1994) algorithms are two major model-free reinforcement learning algorithms.

### 4.3.1 Q-Learning

Q-Learning algorithm is an important off-policy model-free reinforcement learning algorithm for temporal difference learning. It can be proven that given sufficient training under any $\epsilon$-soft policy, the algorithm converges with probability 1 to a close approximation of the action-value function for an arbitrary target policy. Q-Learning learns the optimal policy even when actions are selected according to a more exploratory or even random policy.

The update of state-action values in Q-learning is defined by

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha \left[ r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]. \tag{4.1}$$

The parameters used in the Q-value update process are:

$\alpha$ - the learning rate, set between 0 and 1. Setting it to 0 means that the Q-values are never updated, hence nothing is learned. Setting a high value such as 0.9 means that learning can occur quickly.

$\gamma$ - discount factor, also set between 0 and 1. This models the fact that future rewards are worth less than immediate rewards. Mathematically, the discount factor needs to be set less than 0 for the algorithm to converge.

In this case, the learned action-value function, $Q$, directly approximates $Q^*$, the optimal action-value function, independent of the policy being followed. This dramatically simplifies the analysis of the algorithm and enabled early convergence proofs. The policy still has an effect in that it determines which state-action pairs are visited and updated. However, all that is required for correct convergence is that all pairs continue to be updated. Under this assumption and a variant of the usual stochastic approximation conditions on the sequence of step-size parameters, $Q_t$ has been shown to converge with probability 1 to $Q^*$. The Q-learning algorithm is shown below.

**Algorithm 5** One-step Q-learning algorithm (Watkins, 1989)

Initialize Q(s,a) arbitrarily;
**repeat**(for each episode):
    Initialize $s$;
    **repeat**(for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$;
        Take action $a$, observe $r$, $s'$;
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$;
        $s \leftarrow s'$;
    **until** $s$ is terminal
**until** all episodes end.

### 4.3.2 SARSA

The SARSA algorithm is an on-policy algorithm for TD-Learning. The major difference between it and Q-Learning, is that the maximum reward for the next state is not necessarily used for updating the Q-values. Instead, a new action, and therefore reward, is selected using the same policy that determined the original action. The name SARSA actually comes from the fact that the updates are done using the quintuple $Q(s,a,r,s',a')$. Where: $s$, $a$ are the original state and action, $r$ is the reward observed in the following state and $s'$, $a'$ are the new state-action pair. The Q-value update rule is defined by

$$Q(s_t,a_t) := Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right] \tag{4.2}$$

It is straightforward to design an on-policy control algorithm based on the SARSA prediction method. As in all on-policy methods, we continually estimate $Q^\pi$ for the behavior policy $\pi$, and at the same time change $\pi$ toward greediness with respect to $Q^\pi$. The general algorithm is given as:

As you can see, there are two action selection steps needed, for determining the next state-action pair along with the first. The parameters $\alpha$ and $\gamma$ have the same meaning as they do in Q-Learning.

**Algorithm 6** On-policy SARSA algorithm (Rummery & Niranjan, 1994)

Initialize Q(s,a) arbitrarily;
**repeat**(for each episode):
    Initialize $s$;
    Choose $a$ from $s$ using policy derived from $Q$;
    **repeat**(for each step of episode):
        Take action $a$, observe $r$, $s'$;
        Choose $a'$ from $s'$ using policy derived from $Q$;
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$;
        $s \leftarrow s'; a \leftarrow a'$;
    **until** $s$ is terminal
**until** all episodes end.

### 4.3.3 Function Approximation using Feature-Based Representations

For problems with high dimensionality, the number of states become astronomically large making Q-value table storage infeasible. So the practical implementation of Q-Learning requires using Q-value approximation via generalization of states, i.e., features.

*Features* are functions from states to real numbers that capture important properties of the state. Using a feature representation, we can write a Q-function (or value function) for any state using a few weights:

$$V(s) = \omega_1 f_1(s) + \omega_2 f_2(s) + ... + \omega_n f_n(s)$$
$$Q(s,a) = \omega_1 f_1(s,a) + \omega_2 f_2(s,a) ... + \omega_n f_n(s,a)$$

By this representation, our experience can be summed up in a powerful numbers. However, states may share features but actually be very different in value.

In robotics, it's often usually to use linear or non-linear feature representations to handle the large-scale state-action spaces.

### SARSA with Linear Function Approximation

You can use a linear function of features to approximate the Q-function in SARSA. This algorithm uses the on-policy method SARSA, because the agent's experiences sample the reward from the policy the agent is actually following,

rather than sampling an optimum policy. A number of ways are available to get a feature-based representation of the Q-function. In this section, we use features of both the state and the action to provide features for the linear function.

Suppose $\{f_1, ..., f_n\}$ are numerical features of the state and the action. Thus, $f_i(s, a)$ provides the value for the $i$-th feature for state $s$ and action $a$. These features are typically binary, with domain $[0, 1]$, but they can also be other numerical features. These features will be used to represent the Q-function.

$$Q_\omega(s, a) = \omega_0 + \omega_1 f_1(s, a) + ... + \omega_n f_n(s, a) \tag{4.3}$$

For some tuple of weights, $\omega = [\omega_0, \omega_1, ..., \omega_n]$. Assume that there is an extra feature $f_0$ whose value is always 1, so that $\omega_0$ does not have to be a special case.

An experience in SARSA of the form $\{s, a, r, s', a'\}$ (the agent was in state $s$, did action $a$, and received reward $r$ and ended up in state $s'$, in which it decided to do action $a'$) provides the new estimate of $r + \gamma Q(s', a')$ to update $Q(s, a)$. This experience can be used as a data point for linear regression. Let $\delta = r + \gamma Q(s', a') - Q(s, a)$. Weight $\omega_i$ is updated by

$$\omega_i \longleftarrow \omega_i + \alpha\delta f_i(s, a).$$

This update can then be incorporated into SARSA, giving the algorithm 7.

Similarly, we can use linear function approximation on Q-learning algorithm. The transition is of the form $\{s, a, r, s'\}$, and the difference

$$\delta = \left[r + \gamma \max_{a'} Q(s', a')\right] - Q(s, a).$$

The update rules for Q-values and weights are:

$$\begin{aligned} Q(s, a) &\longleftarrow Q(s, a) + \alpha\delta; \\ \omega_i &\longleftarrow \omega_i + \alpha\delta f_i(s, a). \end{aligned}$$

## 4.4 Neural Network based Q-Learning

In this section we present our contribution that incorporating a neural network to Q-learning. One of the strengths of Q-learning is that it is able to compare

**Algorithm 7** SARSA with linear function approximation (Poole & Mackworth, 2010)

**Input:**
$f = [f_1, ..., f_n]$: a set of features;
$\gamma \in [0, 1]$: discount factor;
$\alpha > 0$: step size for gradient descent.
Initialize weights $\omega = [\omega_0, \omega_1, ..., \omega_n]$;
Observe current state $s$;
Select action $a$ (stochastic policy $\pi(a|s)$);
**repeat**
    Carry out action $a$ (transition probability $Pr(s'|s, a)$);
    Observe state $s'$ and receive reward $r$;
    Select action $a'$ using a policy based on $Q_\omega$ (stochastic policy $\pi(a'|s')$);
    Let $\delta = r + \gamma Q(s', a') - Q(s, a)$;
    **for** $i = 0$ to $n$ **do**
        Update weights: $\omega_i \leftarrow \omega_i + \alpha\delta f_i(s, a)$;
    **end for**
    $s \leftarrow s'$, $a \leftarrow a'$;
**until** Termination.
**Output:** An optimal weights $\omega*$.

the expected utility of the available actions without requiring a model of the environment. Q-learning can handle problems with stochastic transitions and rewards.

### 4.4.1 State and Action Spaces

Since reinforcement learning algorithms can be modeled as an MDP, we need first to define the state space $\mathcal{S}$ and action space $\mathcal{A}$.

Q-learning is aimed at learning a mapping from the state input to the action output. In a navigation problem, the robot perceives the state from the environment by means of its sensors, and this state of environment is used by a reasoning process to determine the action to execute in the given state.

The action space $\mathcal{A}$ is defined by six discrete robot actions. Among them, five are basic moving actions*: move forward (F), turn left at 30° (L30), turn left at 60° (L60), turn right at 30° (R30), turn right at 60° (R60) and one emergency action: move backward (B). The six actions are based on the robot heading

*These five moving actions are defined as the same ones that we used in the previous chapter.

orientation. A turn at 90° is not defined because a sharp turn will bring great danger to a real vehicle. Moving backward is regarded as an emergency action that is only executed when no path is available in front of the robot and whatever turns cannot avoid obstacles. This action is not considered in the self-learning process, because it is, in some sense, an 'instinct' behavior of mobile robots. In sum,

$$\mathcal{A} = \{F, L30, L60, R30, R60, B\}. \tag{4.4}$$

The state space $\mathcal{S} = \{S_1, S_2, S_3, ..., S_N\}$ is composed of a very large but finite number of states, and each state $S_t$ is defined based on the 11-dimensional state representation that we gave in Equation (3.17).

However, we do not consider the degrees of danger of the No.5 and No.6 sensors (see Figure 2.6) thanks to the instinct action 'B'. This can help reduce the complexity of the computation of the dimensionality. Thus, $D_t = [d_1, d_2, d_3, d_4, d_7, d_8, d_9]^\top \in \mathbb{R}^{7\times 1}$.

When a sensor detects an obstacle, a robot should distinguish if this obstacle is an object that can be bypassed or a wall that cannot be bypassed. Hence, we introduce a vector $U_t \in \mathbb{R}^{7\times 1}$ to indicate if a wall is detected ($U = 1$) or an object obstacle ($U = 0$) for the seven sensors in $D_t$.

Therefore, a state can be defined by four groups of features, and is expressed in:

$$S_t = \begin{bmatrix} D_t \\ V_t \\ I_t \\ U_t \end{bmatrix} \in \mathbb{R}^{16\times 1}. \tag{4.5}$$

$V_t \in \{1, 2, ..., 9\}$ is the target region and $I_t \in \{0, 1\}$ is the indicator that determines if the robot has detected the target.

Note: Since the robot needs to deal with different environments, and the robot position coordinates have no direct relation with the target and the obstacles in different environments, it is not necessary to include them as part of the state features. The robot localization can be fixed by many methods and is not the focus of this thesis, so we assume the robot know its position relative to its start position.

### 4.4.2 Reward Function

The reward function measures the immediate feedback for the action taken at a given state. It evaluates how good or how bad the performed action is. Before giving the reward function, one environment state at each time instant is classified into four properties, called the state property $p_t$:

One world state $S_t$ is classified into five properties, called the state property $p_t$:

- Safe State (SS): a state where no surrounding obstacles has been detected.
- Cozy State (CS): a state where the robot has a low or no probability of collision with surrounding obstacles.
- Dangerous State (DS): a state where the robot has a high probability of collision with some obstacles in the environment.
- Winning State (WS): a terminate state where the robot reaches its target.
- Failure State (FS): a terminate state where the robot collides with obstacles.

$$p_t = \begin{cases} SS, & d_{bou} \leq d_{r-o} \\ CS, & d_{cozy} \leq d_{r-o} \leq d_{bou} \\ DS, & d_{col} \leq d_{r-o} \leq d_{cozy} \\ WS, & d_{r-t} \leq d_{win} \\ FS. & d_{r-o} \leq d_{col} \end{cases} \tag{4.6}$$

$d_{r-o}$ and $d_{r-t}$ define the distances between the robot and obstacles, and the target. $d_{bou}$ defines the boundary distance (sensor detection range) of SS and other states. $d_{col}$ defines the radius of the collision region around the obstacle. $d_{win}$ defines the radius of the winning region around the target. $d_{cozy}$ defines the cozy distance. These distances can be found in Figure 2.7.

Based on the state properties, the reward function $r(t)$ is defined in Table 4.1.

$d_{min}^{t-1}$ and $d_{min}^{t}$ are the minimum distances between the robot and the surrounding obstacles respectively at instant $t-1$ and instant $t$. $d_{warn}$ is a warning distance that the robot is approaching too close to an obstacle.

| State Transition | Extra Criteria | $r$ |
|---|---|---|
| other states → Winning State | | 1 |
| Safe State → Cozy State | | 0 |
| Cozy State → Safe State | | 0 |
| Dangerous State → Cozy State | | 0.5 |
| Cozy State → Dangerous State | | 0 |
| Dangerous State → Failure State | | -1 |
| Dangerous State → Dangerous State (approaching obsacles) | $d_{warn} \leq d_{min}^t \leq d_{min}^{t-1} - 2$ | -0.2 |
| | $d_{min}^t = d_{min}^{t-1} - 1 \leq d_{warn}$ | -0.2 |
| | $d_{min}^t = d_{min}^{t-1} - 2 \leq d_{warn}$ | -0.5 |
| | $d_{min}^t \leq d_{min}^{t-1} - 3$ | -1 |
| Nonsafe State → Nonsafe State (evading obstacles) | $d_{warn} \leq d_{min}^t$ | 0.6 |
| | $d_{min}^t \geq d_{warn}$ | 0.4 |

Table 4.1: Reward Function

### 4.4.3 The Stochastic Control Policy

A reinforcement learning agent learns from the consequences of its state-action pairs rather than from being explicitly taught, and it selects its actions on basis of its past experiences and also by new choices. If we may visit each state-action $(s, a)$ a sufficient large number of times, we could obtain the state values via, for example, Monte Carlo methods. However, it is not realistic, and even worse, many state-action pairs would not be visited once. It is important to deal with the exploration-exploitation tradeoff.

In our work, we transplant a Boltzmann distribution to express a stochastic control policy. The learning agent tries out actions probabilistically based on their Q-values. Given a state $s$, the stochastic policy outputs an action $a$ with probability:

$$\pi(s, a) = P(a \mid s) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_{b \in \mathcal{A}} e^{\frac{Q(s,b)}{T}}}. \tag{4.7}$$

where $T$ is the temperature that controls the stochasticity of action selection. If $T$ is high, all the action Q-values tend to be equal, and the agent choose a random action. If $T$ is low, the action Q-values differ and the action with the highest Q-value is preferred to be picked. Thus, $P(a|s) \propto e^{\frac{Q(s,a)}{T}} > 0$ and

$\sum_a P(a|s) = 1$.

We do not fix the temperature to a constant, since random exploration throughout the whole self-learning process takes too long to focus on the best actions.

At the beginning, all $Q(s, a)$ are generated inaccurately, so a high $T$ is set to guarantee the exploration that all actions have a roughly equal chance of being selected. As time goes on, a large amount of random exploration have been done, and the agent could gradually exploit its accumulating knowledge. Thus, the agent decreases $T$, and the actions with the higher Q-values become more and more likely to be picked. Finally, as we assume $Q$ is converging to $Q^*$, $T$ approaches zero (pure exploitation) and we tend to only pick the action with the highest Q-value:

$$P(a|s) = \begin{cases} 1, & \text{if } Q(s,a) = \max_{b \in \mathcal{A}} Q(s,b) \\ 0, & \text{otherwise} \end{cases} \tag{4.8}$$

In sum, the agent starts with high exploration and converts to exploitation as time goes on, so that after a while we are only exploring $(s, a)$'s that have worked out at least moderately well before.

### 4.4.4 State-Action Value Iteration

The $Q$-value function expresses the mapping policy from the perceived state of environment to the executing action. One $Q$-value $Q(s_t, a_t)$ corresponds with one specific state and one action in this state. Autonomous navigation tasks are our main research problems. Like many robotic applications in real life, they have a large-scale state and action spaces. Traditionally, all the state or action values are store in a $Q$-table. However, this is not practical and computationally expensive for large-scale problems. In our method, We propose to predict all state $Q$-values by using a three-layer neural network, as shown in Figure 4.1.

The inputs are the state features that the robot perceives in the surrounding environment, and the outputs correspond to all the action $Q$-values. Therefore, according to Equations (4.4) and (4.5), the network has 16 neurons in the input layer, and 5 in the output layer*. Moreover, 8 neurons are designed in the hidden layer.

*Remember that the action 'B' is an instinctive action that we do not need to learn, so only the other five actions are considered in the output layer.

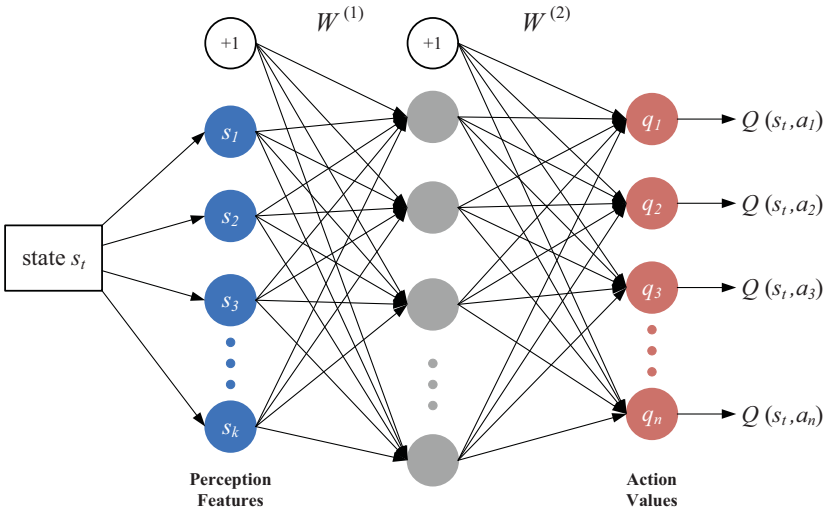



Figure 4.1: A three-layer neural network architecture.

The bias units are set to 1. The weight $W^{(1)} \in \mathbb{R}^{8\times 17}$ is used to connect the input layer and the hidden layer, and similarly, the weight $W^{(2)} \in \mathbb{R}^{5\times 9}$ links the hidden layer and the output layer. The sigmoid function is used for calculating the activations in the hidden and output layers.

We denote $Q(s_t)$ a vector of all action-values in the state $s_t$, and use $Q(s_t, a_t)$ to specify the $Q$-value of taking $a_t$ in $s_t$. Thus,

$$Q(s_t) = \begin{bmatrix} Q(s_t, a_1) \\ Q(s_t, a_2) \\ Q(s_t, a_3) \\ Q(s_t, a_4) \\ Q(s_t, a_5) \end{bmatrix}.$$

The action value iteration is realized by updating the neural network by the means of its weights. In the previous chapter, the neural network was applied for supervised learning where the label for each training state-action pair was explicitly provided. Differently, the neural network in the reinforcement learning does not has label outputs. Q-learning is a process of value iteration and the optimal value after each iteration serves as the target value for neural network training. The update rule is

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha \left[ r_t + \gamma \max_{a \in A} Q_k(s_{t+1}, a) - Q_k(s_t, a_t) \right]. \tag{4.9}$$

where the initial action values $Q_0$ of al the state-action pairs are generated randomly between 0 and 1. $Q_{k+1}(s_t, a_t)$ is treated as the target value of the true value $Q_k(s_t, a_t)$ in the $(k+1)^{\text{th}}$ iteration.

In the vector $Q_k(s_t)$, only $Q_k(s_t, a_t)$ is updated to $Q_{k+1}(s_t, a_t)$, and the rest elements stay unchanged. Sometimes, $Q_{k+1}(s_t, a_t)$ may exceed the range $[0, 1]$, then we need to rescale $Q_{k+1}(s_t)$ to make sure all its components are in $[0, 1]$. We denote $\tilde{Q}_{k+1}(s_t)$ the rescaled action value. To make it clear, the update of $Q$-value is realized along the road $Q_k \to Q_{k+1} \to \tilde{Q}_{k+1}$.

The network error is a vector of form:

$$\delta_{k+1} = \tilde{Q}_{k+1}(s_t) - Q_k(s_t). \tag{4.10}$$

We employ the stochastic gradient descent (SGD) to train the neural network online. The goal is to minimize the cross-entropy cost function $J$ defined as:

$$J = -\left[ \sum_{i=1}^{N_A} (\tilde{Q}_{k+1})_i \cdot \log(Q_k)_i + (1 - (\tilde{Q}_{k+1})_i)(1 - \log(Q_k)_i) \right]. \tag{4.11}$$

where $N_A$ is the number of actions used for training. In our navigation tasks, $N_A = 5$.

The action $Q$-values are nonlinear functions of weights of the network. SGD optimizes $J$ and updates weights by using one or a few training examples according to:

$$W^{(i)} \leftarrow W^{(i)} - \sigma \frac{\partial J}{\partial W^{(i)}}. \tag{4.12}$$

Each iteration outputs new weights $W^{(i)}$ and a new cost $J'$ is calculated. This update repeats until it arrives at a maximum times of iteration or $|J' - J| < \epsilon$.

### 4.4.5 Algorithm

An autonomous navigation tasks via NNQL can be divided into two processes. The first one is the training process to endow the robot with the self-learning ability, and the second one is the navigation process to use the trained policy to execute an independent navigation task (Xia & El Kamel, 2015d).

#### 4.4.5.1 Training Process of NNQL

Training the mobile robot is done by exposing it to a bunch of learning episodes and each episode has a different environment. Also, the robot has a set of initial starting positions. The variety helps the robot to encounter as many situations as possible, which could accelerate the learning speed.

Each episode starts by perceiving the current state of the environment. The robot detects the surrounding obstacle locations through its sensors, and a rough target region* is supplied to the robot. Once the current state is checked, if it is a Safe State, the robot executes a goal-oriented behavior that it changes its orientation towards the target position, and moves one step forward trying to get closer. If the current state is Non-Safe State, the robot sends the current state features into the FFNN and outputs all the possible $Q$-values. According to the action selection strategy represented by a stochastic control policy, the robot picks an action and moves to a new state. Then, the robot checks the resulting new state, receives the immediate reward and updates the $Q$-values accordingly. Then the updated $Q$-values are sent back to the neural network and use the backpropagation algorithm to update the weights of the network.

Each episode has a limited amount of moving steps. The robot needs to reach the target within the steps. If the robot runs out of the steps and does not reach the target, or if the robot collides with an obstacle or reach the target, the episode is terminated and a new episode is started.

The key of training efficiency is greatly related to how to make use of the accumulated sequence of state-action pairs and their $Q$-values. A bunch of previous work (Jaradat *et al.*, 2011; Qiao *et al.*, 2008; Xia & El Kamel, 2015d) used one-step Q-learning to update one $Q$-value at a time. When the robot arrives at

*"A rough target region" means that the robot only knows in which sensor region locates the target, but has no idea of the exact direction and the relative distance between itself and the target. This concept was also presented in Equation (4.5).

a new state, only the new $Q$-value will be updated and the previous action values will be discarded. Others used batch learning (Riedmiller, 2005) that updates all the $Q$-values once they are all collected. This also poses some advantages. First, without online update, we cannot guarantee that the collected $Q$-values have their optimal target values. Moreover, waiting all the values being obtained is always time-wasting. We propose to update online not only the current $Q$-value but also gather the previous values to train together.

The training algorithm is given in Algorithm 8.

**Algorithm 8** Training algorithm of NNQL

Initialize the NN weights $W^{(1)}$ and $W^{(2)}$ randomly;
**for** all episodes **do**
    Give the target and generate obstacles randomly;
    Load the robot initial position $[x_0, y_0]$ and orientation $\theta_0$;
    Observe current state $s_1$ and state property $p_1$;
    $input := [\,]$, $target := [\,]$;
    $t \leftarrow 1$;
    **for** all moving steps **do**
        Compute all action-values $\{Q(s_t, a_i)\}_i$ in state $s_t$ via NN;
        Select one action $a_t$ according to the stochastic policy $\pi(s, a)$ in (4.7), and then move;
        Observe new state $s_{t+1}$ and state property $p_{t+1}$;
        Obtain the immediate reward $r_t$;
        Update the Q-value function from $Q(s_t, a_t)$ to $\tilde{Q}(s_t, a_t)$ via (4.9);
        Apply feature scaling for $\tilde{Q}$ to the range $[0, 1]$;
        Add $s_t$ to $input$, and add $\tilde{Q}(s_t, a)$ to $target$;
        Apply SGD to train $(input, target)$ and to update the weights $W^{(1)}$ and $W^{(2)}$;
        **if** $s_{t+1}$ is Winning State or Failure State **then**
            Start a new episode;
        **end if**
        $t \leftarrow t + 1$;
    **end for**
**end for**

#### 4.4.5.2 Robot Navigation Using NNQL

After training the robot, the resulting policy is still stochastic but near-deterministic that used by the robot for future navigation tasks in various environments.

The robot starts its navigation through the environment by finding its current state. If it is a Safe State the robot does not need to follow the policy but changes its orientation towards the target and moves one step forward. It continues moving until entering a Non-Safe region where the robot needs to adopt the trained control policy. The robot uses the FFNN to generate all possible state-action $Q$-values. The robot greedily takes the action that has the biggest $Q$-value. After that, the robot finds its new state and repeats the process of action selection until the robot reaches its goal or collide an obstacle. The navigation algorithm is shown in Algorithm 9.

**Algorithm 9** Robot Navigation using NNQL

```
Load the trained NN weights W^(1) and W^(2);
Give the target and generate obstacles randomly;
Load the robot initial position [x_0, y_0] and orientation θ_0;
t ← 1;
for all moving steps do
    Observe current state s_t and state property p_t;
    if s_t is Winning State or Failure State then
        Terminate the navigation;
    end if
    Compute all action Q-values {Q(s_t, a_i)}_i via neural network;
    Pick the moving action a_t according to greedy policy, and then move;
end for
```

## 4.5 Experimental Results

We employ the mobile robot model described in Chapter 2.5 to undertake autonomous navigation tasks in unknown environments.

The environment map has a size of $100 \times 100$ m$^2$. The obstacles are randomly scattered in the environment and the robot has no prior knowledge of their numbers, sizes and positions. The initial positions of the robot (a blue solid rectangle in the map) is placed at $(10, 10)$ or $(90, 10)$. The target positions (a red hollow circle in the map) may be found at $(90, 90)$ or $(10, 90)$. The velocity of the robot is fixed at 2.5 m/s. The mission of the robot is to start from the initial position and to find an optimal path to arrive at the target position without any collision

with any obstacles. If no obstacles are detected, the robot is designed to stick to a goal-oriented behavior.

We use a 16-8-5 three-layer neural network to approximate all action $Q$-values. It is also used to generalize to unvisited states.

### 4.5.1 Self-learning Results

The self-learning process is conducted in a number of learning episodes, and each episode has a maximum of 400 moving steps. All episodes have different configurations of random obstacle positions. A new episode will be started in the following three situations:

- The robot finds a collision free path to the target;
- The robot collides with an obstacle or the map borders;
- The robot runs out the moving steps.

The training process will be terminated when all the learning episodes are finished. The experiment parameters are selected as follows:

- Learning rate of $Q$-function in Equation (4.9): $\alpha = 0.8$,
- Discount factor of $Q$-function: $\gamma = 0.7$,
- Learning rate of SGD in Equation (4.12): $\sigma = 0.3$,
- Maximum temperature in Equation (4.7): $T_{\max} = 0.9$,
- Minimum temperature: $T_{\min} = 0.9$.

The learning results are shown in Figure 4.2.

300 episodes are set for the robot to learn to navigate in different environments. One advantage of learning in different environments is that the robot can always face new challenge and avoid falling into infinite loops when it gets used of its environment. We can see that the robot tried different actions and the rewards evaluated this action decision, which can help the robot to correct the action selection in the future. During all episodes, the proposed algorithm adopted BPNN to train the weights $W^{(1)}$ and $W^{(2)}$. When the learning process is completed, the

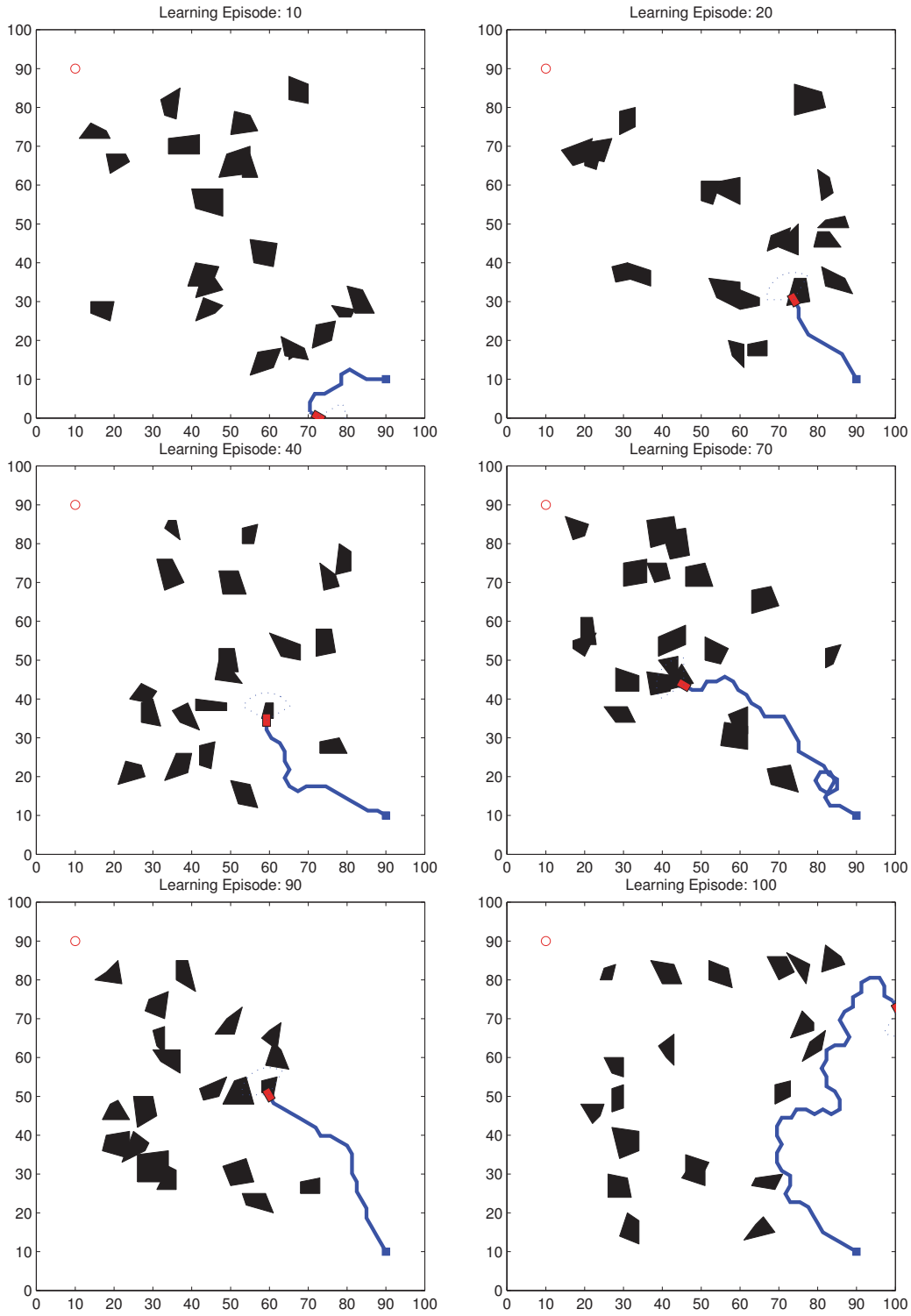
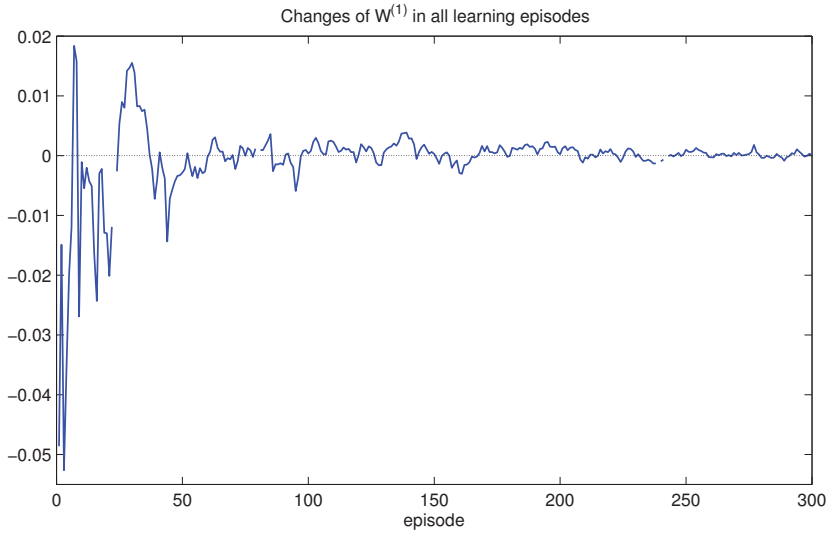
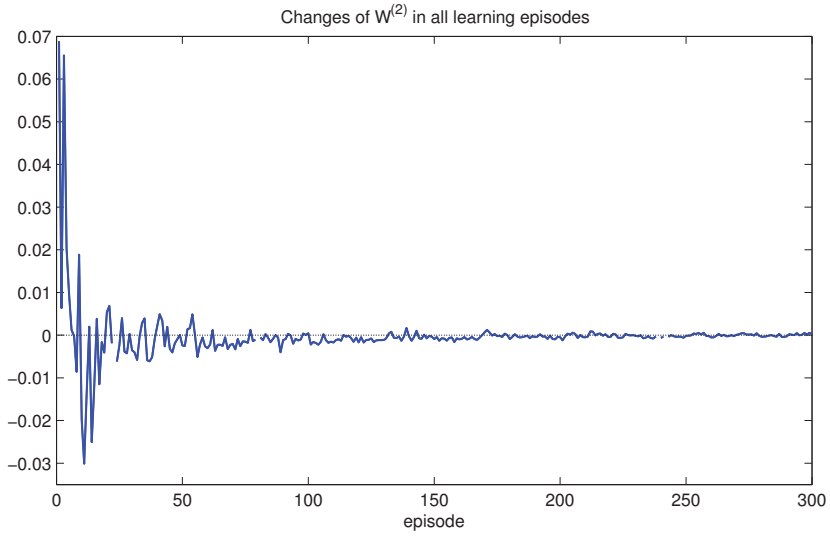



Figure 4.2: Learning results in different episodes via NNQL.

weights have been well trained and can be used directly in the following process of robot navigation. Since the proposed algorithm is a trial-and-error method, it is reasonable that a great number of episodes failed due to a collision.




(a) Changes of $W^{(1)}$ in all learning episodes




(b) Changes of $W^{(2)}$ in all learning episodes

Figure 4.3: Changes of NN weights in all learning episodes.

We measured the changes of the weights of the network in Figure 4.3. The change in one episode is computed by the difference between the weight at the beginning of the episode $\tilde{W}^{(i)}$ and the weight at the end of the episode $\tilde{\tilde{W}}^{(i)}$:

$$\Delta W^{(i)} = \sum_j \sum_k \left( \tilde{\tilde{w}}_{j,k}^{(i)} - \tilde{w}_{j,k}^{(i)} \right).$$

The figures showed that both of the weights fluctuated strong in the beginning episodes and then went quiet around 0 after 100 episodes. The overall trend of weight changes was turning stable as the learning process carried on and the weights converged to their optimal values. That is to say, the robot has gradually learned how to cope with the surrounding environment.

The numbers of successful learning in every 100 episodes is shown in Figure 4.4.

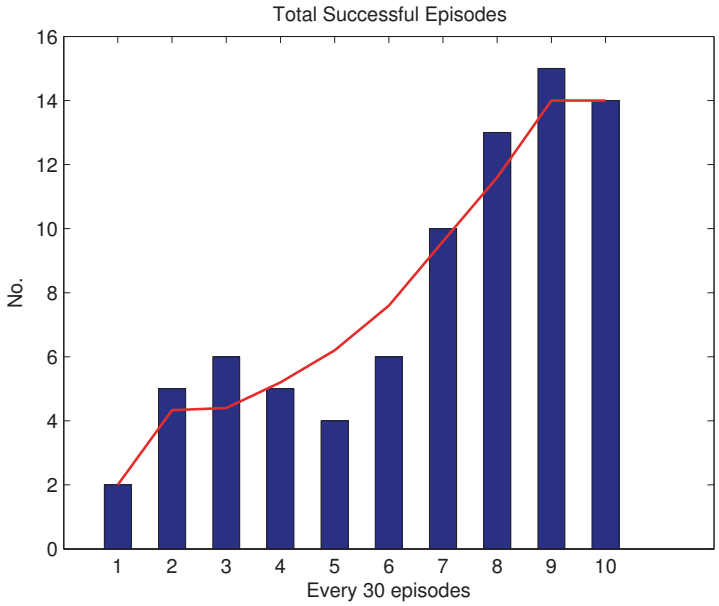



Figure 4.4: Numbers of successful learning episodes in every 30 episodes.

A successful learning is an episode in which the robot succeeds in arriving at the target position. In the first 100 episodes, less than 10 were successful, but in the last 100 episodes, more than 60 witnessed the robot arriving at the target. This remarkably increasing trend demonstrated that the proposed NNQL algorithm helped the robot to become intelligent.

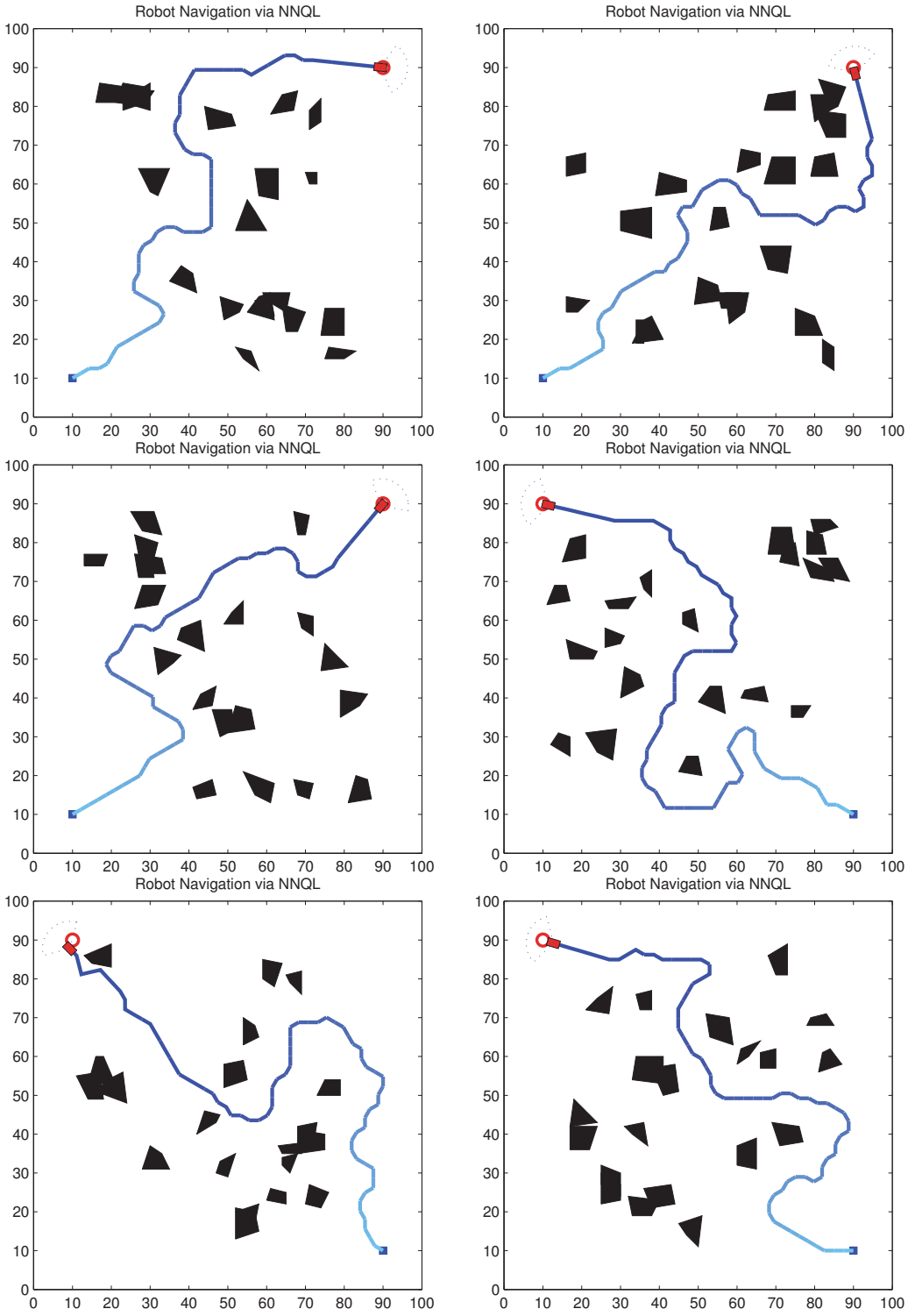



Figure 4.5: Autonomous navigation results in different environments via NNQL.

### 4.5.2 Autonomous Navigation Results

In the navigation process, the weights $W^{(1)}$ and $W^{(2)}$ have converged to their optimal values. The robot has learned how to behave in front of obstacles. The FFNN is now only adopted, and the robot chooses the best fit action. Then it is the time for the mobile robot to demonstrate its intelligence of independent navigation in an unexpected environment.

Figure 4.5 showed six completely new navigation missions using the same weights. The robot tried from different start points to reach different target positions. The robot succeeded in getting to the target position in a collision-free path in all four environments and the robot always kept a safe distance to the surrounding obstacles.

It is observed that the paths that the robot chose may not be the optimal ones due to a lack of complete knowledge of environment map, but they are still fully acceptable and perfect enough to meet our expectation, especially in such complicated environments.

In order to test the robustness of the algorithm, we tried six different robot speeds and for each speed we conducted 100 experiments. The results are shown in Table 4.2.

| Robot speed (m/s) | 1.5 | 2 | 2.5 | 3 | 3.5 | 4 |
|---|---|---|---|---|---|---|
| Rate of success (%) | 100 | 100 | 100 | 97 | 94 | 90 |

Table 4.2: Compassion of the rate of success versus different robot speeds.

It is normal that a higher speed means a shorter reaction time, and thus a stronger possibility of collision. These tests with different speeds achieved a high rate of success.

Therefore, the above experiments have proven the feasibility and the stability of the proposed NNQL algorithm.

### 4.5.3 Comparison and Analysis

For our study, we implement and compare several methods: the first one is our proposed method NNQL; the second one is the online one-step Q-learning (online QL) that adopted in many previous work; the third one is the neural fitted-Q

iteration (NFQ), which is a successful batch learning method. We conducted 50 times for each method.

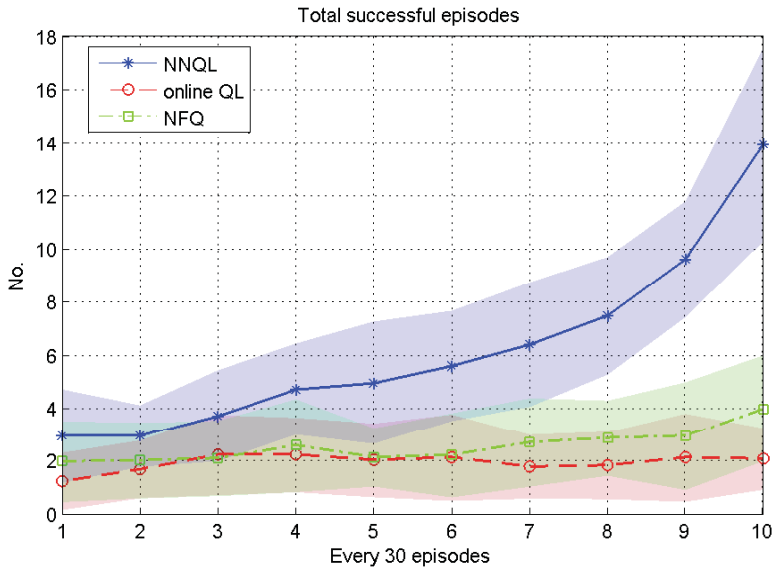



Figure 4.6: Numbers of successful learning episodes in every 30 episodes.

Figure 4.6 showed the comparison of total successful learning in every 30 episodes. The blue solid line is the average result of NNQL over 50 experiments, the red dashed line is the online QL method, and the green dash-dot line is NFQ algorithm. The shaded areas represent the standard derivation. We can see that NNQL outperformed the other two popular methods. NFQ did not begin training until all the training data had been collected, and hence it was easy to fail an episode without adjusting its control policy. Online QL did the worst since this 'extravagant' method abandoned so many training data and 300 episode was far from enough.

Let the mobile robot navigate autonomously in unknown environments after being trained by those three methods. We fixed the robot velocity at 2.5 m/s and a total amount of 300 learning episodes. For each method we obtained 50 control policies and for each policy we carried out 100 navigation tasks. The results were compared in Table 4.3.

We can tell that our method achieved the best result, NFQ did slightly worse, and online QL needed more episodes to catch up with the performance.

| | NNQL | Online QL | NFQ |
|---|---|---|---|
| Average rate (%) | 98 | 56 | 87 |
| Best rate (%) | 100 | 74 | 97 |
| Worst rate (%) | 94 | 29 | 83 |

Table 4.3: Comparison of the rate of success versus different methods.

### 4.5.4 Autonomous Navigation in Dynamic Environments

We tested the NNQL algorithm in dynamic environments. The results are shown in Figure 4.7. In the environment, the black solid objects are static obstacles, and the purple rectangle ones are moving obstacles.

At time 31, the robot met some static obstacles around him, and successfully avoid them. At time 50, before arriving at the destination, the robot found a moving obstacle in front of him, and the robot successfully avoided it and finally arrived at the destination. Therefore, NNQL has proven its feasibility in the application of autonomous navigation in dynamic environments.

### 4.5.5 Discussions

From the above experiment results, NNQL has demonstrated good and reliable performance in autonomous navigation tasks without explicitly programming robot behaviors. With 300 episodes, which is not a big number, the robot can execute an independent navigation with almost 100% confidence. Compared with two popular model-free algorithms, NNQL outperformed them. We can safely reach the conclusion that NNQL has successfully empowered a mobile robot with strong self-learning ability in autonomous navigation tasks.

One limitation is that under stochastic policies, the path that the robot choose may not be optimal, and the robot may wonder for a while in some place before finding a way out. That is because the robot do not have a global view of the environment and therefore has to explore the environment in order to memorize a small range of the surrounding environment. One potential solution is to incorporate simultaneous localization and mapping (SLAM) techniques.

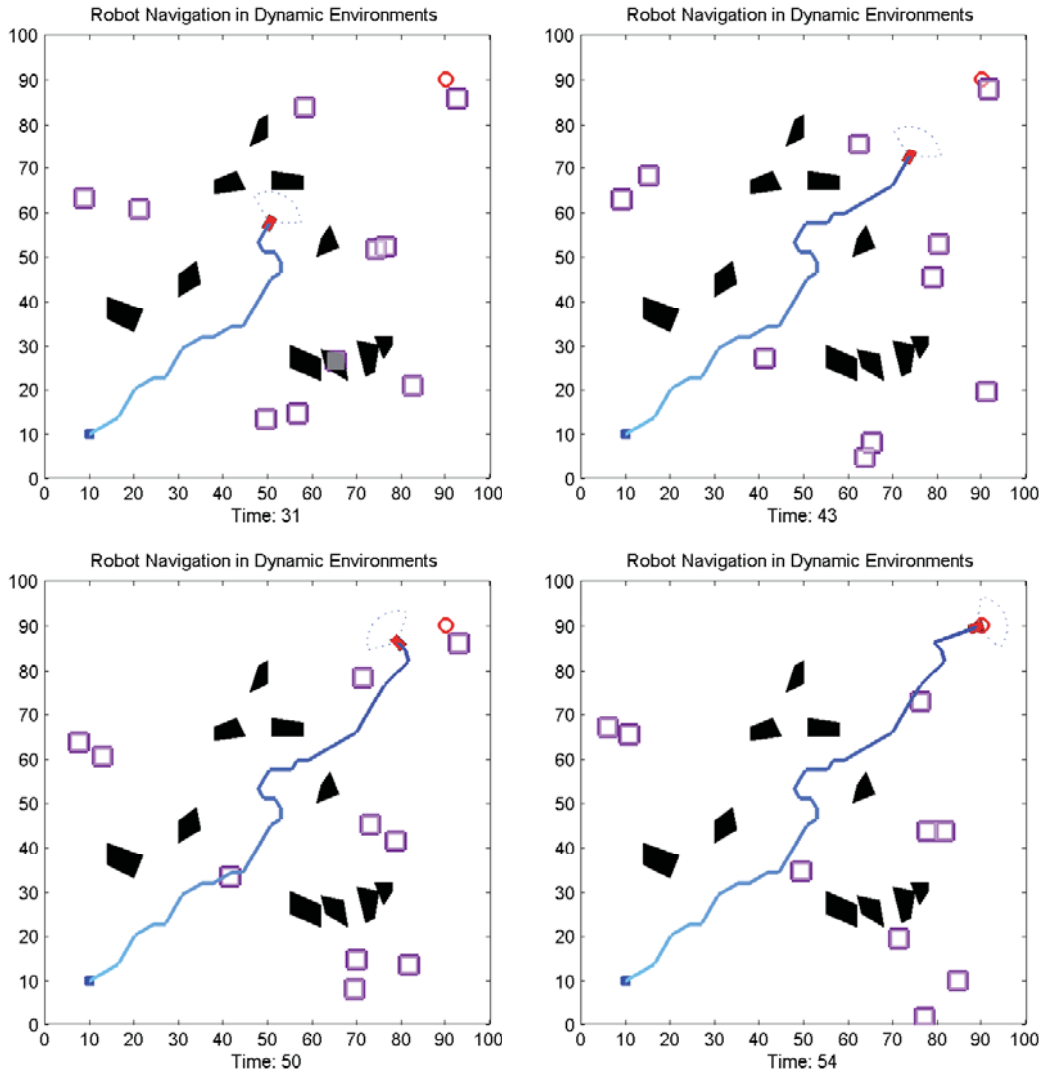



Figure 4.7: Autonomous robot navigation in a dynamic environment using NNQL.

## 4.6 Conclusion

Much of classical robotics focused on reasoning, optimal control and sensor processing given models of the robot and its environment. While this approach is successful for many industrial applications, it falls behind the more ambitious goal of robotics as a test platform for our understanding of artificial intelligence.

This chapter presented the robot self-learning strategy without prior experience under explicit feedback. We explored the mobile robot navigation problem by combining reinforcement learning and neural network. Q-learning is applied to enhance the self-learning ability of a mobile robot through trial-and-error interactions with an unknown environment. We designed new reward expression and introduced the neural network architecture to store and train the large-scale Q-values, and also to generalize the learning performance to large-scale state and action spaces. The experiment results show the feasibility and the stability of the proposed method. The robot can complete navigation tasks safely in an unpredicted dynamic environment and becomes a truly intelligent system with strong self-learning and adaptive abilities.

# Chapter 5

# Learning Reward Functions with Nonlinear Neural Policy Representations

## Contents

## 5.1 Introduction

In this day and age, more and more robots are produced to assist or replace our human beings to perform complicated planning and control operations and tasks, such as manipulating objects, navigating in outdoor environments, and driving in urban areas. We are aware of the fact that designing such controllers for these tasks is usually a complicated process, even for people specialized in programming robots, which requires creating by hand a new and different controller for each particular task. The designer has to take deliberately the wide range of situations that the robot may face into account. This sort of manually programming is generally an expensive as well as intense time-consuming process. Rather than pre-programming a robot for all the tasks, it would be more useful if the robot could learn such tasks by themselves.

In the previous chapter, we presented our reinforcement learning technique. In this framework, an learning robot interacts with a dynamic environment and finds a policy through a reward function. This method was proven to be a powerful and reliable solution to robot learning. However, one fundamental assumption of successful RL algorithms is the reward function, the most succinct representation of the designer's intention, needs to be provided beforehand. In practice, defining the reward function can itself be a challenge because an informative reward function may be very hard to specify and exhaustive to tune for large and complex problems. The reward function we used in the previous chapter was not quite a straightforward one.

People are wondering if we could extract a reward function from demonstrated examples of a desired behavior, and this inspires the advent and the development of Inverse Reinforcement Learning (IRL). IRL can be taken as an extension of reinforcement learning, which directly tackles the problems of learning a reward function through expert demonstrations. It was originally introduced in (Ng & Russell, 2000), where the authors provided a formal characterization of the solution space for the IRL problem and three categories of algorithms to tackle different tasks. In the sequel, many algorithms have been proposed in the subject of IRL (Abbeel & Ng, 2004; Neu & Szepesvári, 2007; Ramachandran & Amir, 2007; Ratliff *et al.*, 2006; Ziebart *et al.*, 2008).

Inverse reinforcement learning is a particular case of learning from demonstrations. Therefore, generalizing the rewards on undemonstrated states is crucial since the demonstrations cannot cover all the states, such as in an autonomous robot navigation problem where the state space is always of large scale. In order to allow for generalization, the cost function takes as input a set of features that describe the current state or have been extracted from the current perceptions, rather than these raw inputs themselves. Current inverse reinforcement learning algorithms generally solve this problem by using a linear combination of state and action features to generalize on undemonstrated states and do not give an explicit policy representation in large-scale spaces.

Our work focuses on improving autonomous robot navigation via inverse reinforcement learning. We adopt neural network to generalize the expert's actions to unvisited regions of the state space and an explicit policy representation is easily expressed by neural network. Hence, we proposed an efficient and convenient algorithm called the Neural Inverse Reinforcement Learning (NIRL) (Xia & El Kamel, 2015a,b) and apply it to autonomous navigation tasks. For computational simplicity, maximum a posteriori (MAP) estimation can be applied, rather than the full Bayesian inference, as also suggested in (Choi & Kim, 2011b). We also investigate the robustness of our algorithm by tuning the parameters, including the features, on a fixed set of demonstrations and then testing their performance on a large sample of demonstrations drawn from some distribution. Experimental results on simulated autonomous navigation problems show that a mobile robot using our approach can successfully navigate to the target position without colliding with unpredicted obstacles, largely reduce the learning time,

and has a good generalization performance on undemonstrated states. Hence we prove the robot intelligence of autonomous navigation transplanted from limited demonstrations to completely unknown tasks.

## 5.2 Related Work

Inverse reinforcement learning can be formulated in Markov decision processes. IRL assumes that an expert demonstrates a task to the learning agent and acts optimally with respect to an unknown reward function to be discovered, and IRL algorithms take these demonstrations as input. It is generally assumed that a model of the transition is known and that the expert is acting optimally or close to optimally with respect to an unknown reward function. The goal of inverse reinforcement learning is then to find a reward for which the expert behavior is optimal.

Inverse reinforcement learning was originally introduced in (Ng & Russell, 2000) where the authors addressed three learning problems: IRL in finite state spaces, IRL in infinite state spaces and IRL from sampled trajectories. In practice, it is easier to get samples from an expert. However, the authors also noted that the IRL problem is ill-posed. In fact, there exists a series of reward functions, including constant functions, that may lead to the same optimal policy. Most existing IRL algorithms are then focused on solving this particular problem by making some assumptions about the form of the demonstrator's reward function.

Abbeel & Ng (2004) introduced a new indirect learning approach, named apprenticeship learning, where the learning is less concerned about the actual reward function, and the objective is to recover a policy that is close to the demonstrated behavior. It is assumed that the reward is a sum of weighted state features, and finds a reward function to match the demonstrator's feature expectations. This method may not explicitly recover the expert's reward function, but still output a policy that attains the performance close to that of the expert.

The maximum margin planning algorithm (Ratliff *et al.*, 2006) uses similar ideas, a linear-in-features reward, where the learner attempts to find a policy that make the provided demonstrations look better than other policies by a margin, and minimizes a cost function between observed and predicted actions by a subgradient descent.

The policy of an agent is derived according to the reward function through RL algorithms, thus a slight change of the reward function can lead to a change in the policy. Gradient methods is used in (Neu & Szepesvári, 2007) where the deviations from the expert's trajectory are penalized through natural gradient.

The Bayesian inverse reinforcement learning approaches (Lopes *et al.*, 2009; Ramachandran & Amir, 2007) use probability distribution to tackle with the ill-posed problem. They assume that the demonstrator samples state-action sequences from a prior distribution over possible reward functions, and calculates a posterior on the reward function using Bayesian inference.

Similar to Bayesian IRL, the maximum entropy algorithm (Ziebart *et al.*, 2008, 2010) use an MDP model for calculating a probability distributionon the state-actions. Maximum entropy IRL focuses on the distribution over trajectories rather than pure actions. Similar research using linear programming is proposed in (Syed *et al.*, 2008). Later on, based on the maximum entropy framework, the relative entropy inverse reinforcement learning algorithm using policy iteration is proposed (Boularias *et al.*, 2011). It indirectly employs knowledge of the environment and minimizes the relative entropy between the empirical distribution of the state-action trajectories under a baseline policy and the distribution of the trajectories under a policy that matches the reward features of the demonstrations. A stochastic gradient descent is used to minimize the relative entropy.

Qiao & Beling (2011) proposes a Gaussian processes model and use preference graphs to represent observations of decision trajectories. Levine *et al.* (2011) present a probabilistic algorithm for nonlinear inverse reinforcement learning and they use Gaussian process model to learn the reward as a nonlinear function.

Much previous work employs a linear approximation with feature expectations. The general idea is to find a reward function minimizing some loss between the expert features expectations and the one of the optimal policy for the approximated dynamics and reward function. Structured Classification for Inverse Reinforcement Learning (SCIRL) (Klein *et al.*, 2012a,b; Piot *et al.*, 2013) uses the quadratic programming maximum-margin approach proposed in (Abbeel *et al.*, 2008) to obtain a reward function and least squares temporal differences (LSTD) (Bradtke & Barto, 1996) for policy evaluation. The subgradient is estimated using importance sampling on trajectories generated using a stochastic

policy on the real environment. LSTD is also used to estimate the feature expectation of the expert. A similar approach is taken in (Mori *et al.*, 2011), which used the game-theoretic algorithm MWAL (Syed & Schapire, 2007) where a near optimal policy is found by using LSPIf, a variant of LSPI (Lagoudakis & Parr, 2003) for discrete finite horizon Markov decision processes.

Another different approach is taken by Levine *et al.* (2010), which constructs reward features from a set of component features through logical conjunctions. Given a set of trajectories, and the true environment model, the algorithm then estimates a reward function by finding the most effective feature combinations.

Inverse reinforcement learning also gets much attention as an effective technique for robot learning (Abbeel *et al.*, 2010; Choi & Kim, 2012; Michini *et al.*, 2013). Other methods also show significant impact in robot intelligence (Cobo *et al.*, 2014; Shon *et al.*, 2007; Silver *et al.*, 2008, 2012).

## 5.3 Inverse Reinforcement Learning

### 5.3.1 Preliminaries

Inverse reinforcement Learning addresses the problem of recovering the unknown reward function for a given Markov decision problem for which the optimal policy matches the examples. The reward function can be used to recover a policy for the expert, and the features can be used to transplant the reward to any novel environment on which the component features are well defined.

Inverse reinforcement Learning is formulated within the framework of a Markov decision process without a reward function, denoted by MDP\R. Formally, a finite-state, infinite horizon MDP is defined as a tuple $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, T, R, \gamma\}$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions. $T$ is a transition function where $T(s, a, s') = P(s_{t+1} = s' \,|\, s_t = s, a_t = a)$ gives the transition probability from state $s$ to state $s'$ when action $a$ is taken. $R$ stands for the reward function where $R(s, a)$ denotes the immediate reward incurred when action $a$ is executed in state $s$. $\gamma \in [0, 1)$ is a discount factor.

We assume that both the agent policy $\pi$ and the environment are Markovian. Consequently, the action $a_t$ only depends on the current state $s_t$ and the next state $s_{t+1}$ only depends on $s_t$, $a_t$.

A policy is a mapping $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, where $\pi(s, a)$ is the probability of choosing action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$.

For a fixed policy, the value function $V^\pi : \mathcal{S} \to \mathbb{R}$ under the policy $\pi$ is defined by

$$V^\pi(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \,|\, s_0 = s \right],$$

where $(s_t, a_t)_{t \geq 0}$ is the sequence of random state-action pairs generated by executing the policy $\pi$. The associated action-value function $Q^\pi(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ under the policy $\pi$ is defined by

$$Q^\pi(s, a) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \,|\, s_0 = s, a_0 = a \right], \tag{5.1}$$

The purpose of the agent is to find an optimal policy $\pi^*$ that maximizes the expected total discounted reward over all states. The optimal value function is defined by $V^*(s) = \sup_\pi V^\pi(s)$, while the optimal action-value function is defined by $Q^*(s, a) = \sup_\pi Q^\pi(s, a)$.

### 5.3.2 Inverse Reinforcement Learning

Inverse reinforcement learning is defined in MDP\R. The agent observes a set of expert demonstrations in the form of $M$ state-action pairs:

$$\mathcal{D} = \{(s_1, a_1), (s_2, a_2), ..., (s_M, a_M)\},$$

where each pair $D_i = (s_i, a_i)$ indicates that the demonstrator took action $a_i$ in state $s_i$ by adhering to some expert policy $\pi^E$. We assume that the expert acts near-optimally trying to solve a certain task.

Inverse reinforcement learning algorithms attempt to find a reward function $R$ whose corresponding optimal policy $\pi^*$ matches the actions taken in the observations $\mathcal{D}$. In our autonomous navigation tasks, the expert demonstrations are composed of $N_T$ trajectories: $\mathcal{D} = \{\tau^{(i)}\}_{1 \leq i \leq N_T}$. $\mathcal{L} = \{l^{(i)}\}_{1 \leq i \leq N_T}$ are the number of state-action pairs contained in each trajectory, satisfying $M = \sum_{i=1}^{N_T} l^{(i)}$, and $l^{(i)}$ may vary according to different trajectories.

We assume that the (unknown) reward function $R$ is a linear combination of $K$ feature vectors $f_k$ with weights $\theta_k$,

$$\forall (s,a) \in \mathcal{S} \times \mathcal{A} : \quad R(s,a) = \theta^\top f(s,a) = \sum_{k=1}^{K} \theta_k f_k(s,a). \tag{5.2}$$

The features indirectly represent the perceived state of environment, and also act as a portable explanation for the expert's policy, enabling the expert's behavior to be predicted in unfamiliar surroundings.

Then the Equation (5.1) can be rewritten as:

$$\begin{aligned} Q^\pi(s,a) &= \theta^\top \mu^\pi(s,a), \\ \text{with } \mu^\pi(s,a) &= \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t f(s_t,a_t) \,|\, s_0 = s, a_0 = a \right]. \end{aligned} \tag{5.3}$$

The term $\mu^\pi$ is called the feature expectations of the policy $\pi$, and they completely determine the expected sum of discounted rewards for acting according to that policy (Abbeel & Ng, 2004). The expectation of one feature $f_i^\pi$ under the policy $\pi$ can thus be expressed by:

$$\mu_i^\pi(s,a) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} \gamma^t f_i(s_t,a_t) \,|\, s_0 = s, a_0 = a \right].$$

If two policies share the same feature expectations, then they will have the same value function whatever reward $R = \theta^\top f$:

$$\mu^{\pi_1} = \mu^{\pi_2} \Longrightarrow \theta^\top \mu^{\pi_1} = \theta^\top \mu^{\pi_2} \Longrightarrow Q^{\pi_1} = Q^{\pi_2}. \tag{5.4}$$

Therefore, inverse reinforcement learning can be transformed to minimizing a distance between the feature expectations of the expert $\mu^E$ and those of the learning agent $\mu^\pi$.

It is well known that the IRL problem is ill-posed. Indeed, $\hat{R}(s) = c, \forall s \in S$, where $c$ is any constant, will make any set of state-action pairs $\mathcal{D}$ trivially optimal. Also, $\mathcal{D}$ may contain inconsistent or conflicting state-action pairs, i.e. $(s_i, a_1)$ and $(s_i, a_2)$ where $a_1 \neq a_2$. Furthermore, the rationality of the demonstrator is not well-defined, i.e., is the demonstrator perfectly optimal, or if not, to what extent sub-optimal.

## 5.4 Nonlinear Neural Policy Representations

A policy connects every state with a corresponding action. If we are dealing with a large-scale state space, an explicit policy representation would be hard to describe, and most of previous work omit the policy representation. With a perfect policy representation, the learning process could be much more efficient.

An artificial neural network can approximate any functions in any accuracy and has a good generalization performance. Thus, we propose an explicit policy representation by incorporating the neural network framework, and this representation can ease the learning by IRL.

### 5.4.1 State and Action Spaces

In order to represent a state of environment, we first denote a numerical danger level of 0 to 7 for each sensor reading, as we defined in the previous chapters. The bigger the level is, the more possible the robot collides with an obstacle. A level of 0 means no obstacle detected in this sensor region and 7 means a collision. Only seven sensors excluding $S5$ and $S6$ are used in the state representation, called state sensors. The reason why we eliminate them is that the experimental results showed a better performance if not considered $S5$ and $S6$ as features.

One state is defined by 16 features: $\phi(s) = \{\phi_i\}_{1 \leq i \leq 16}$, where $(\phi_i)_{\{1 \leq i \leq 7\}} = \{0, ..., 7\}$ are the danger levels of state sensors, $\phi_8 = \{0, 1\}$ represents whether the target can be detected, $\phi_9 = \{0, .., 9\}$ represents the target region, and $(\phi_i)_{\{10 \leq i \leq 16\}} = \{0, 1\}$ represents whether the state sensors detect a wall.

Since the robot needs to deal with different environments, and the robot position coordinates have no direct relation with the target and the obstacles in different environments, it is not necessary to include them as part of the state features. The robot localization can be fixed by many methods and is not the focus of this paper, so we assume the robot know its position relative to its start position.

The action spaces are defined by six robot actions:

$$\mathcal{A} = \{u_1, u_2, u_3, u_4, u_5, u_6\}.$$

Among them, five are basic moving actions: move forward ($u_1$), turn left at 30° ($u_2$), turn left at 60° ($u_3$), turn right at 30° ($u_4$), turn right at 60° ($u_5$) and one

emergency action: move backward ($u_6$). The six actions are based on the robot orientation. A turn at 90° is not defined because a sharp turn will bring danger to a real vehicle. Moving backward is regarded as an emergency action that is only executed when no path is available in front of the robot and whatever turns cannot avoid obstacles.

### 5.4.2 Neural Policy Representation

In general, given a set of $n$ actions , one state at time instant $t$ has a $k$-feature representation $\phi(s_t) = \{\phi_i\}_{1 \leq i \leq k}$*. We can then extend the state representation by a state-action feature representation $f(s_t, a_t)$ at time instant $t$. It is a row vector of size $k \times n$:

$$f(s_t, a_t) = \begin{bmatrix} \varphi(s_t, u_1)^\top & \varphi(s_t, u_2)^\top & \dots & \varphi(s_t, u_n)^\top \end{bmatrix}. \tag{5.5}$$

Here,

$$\varphi(s_t, u_i) = \begin{cases} \phi(s_t), & \text{if } a_t = u_i \\ 0, & \text{otherwise.} \end{cases} \tag{5.6}$$

Then, we denote a function $\mathcal{G} : \mathcal{S} \to \mathbb{R}^{n \times 1}$ a three-layer neural network to link each state $s_t$ with all its corresponding action values $Q(s_t, u)$, i.e.,

$$\mathcal{G}(s_t) = \begin{bmatrix} Q(s_t, u_1) \\ Q(s_t, u_2) \\ \vdots \\ Q(s_t, u_n) \end{bmatrix}, \tag{5.7}$$

where $n = |\mathcal{A}|$. The architecture of the neural policy representation is shown in Figure 5.1.

Generally, the state features $\phi(s)$ is sent to the input layer and the network outputs a vector of action values. Thus, the network has $k$ input units, $L_h$ hidden units and $n$ output units. From now on, we consider the bias units as parts of the input and hidden layers, and they are set to 1. Hence, the input of the network is now $x = \{\phi_i\}_{0 \leq i \leq k}$ where $\phi_0 = 1$ is the bias unit, and the same goes for the

*From now on, we write $s_t$ simply to represent $\phi(s_t)$.

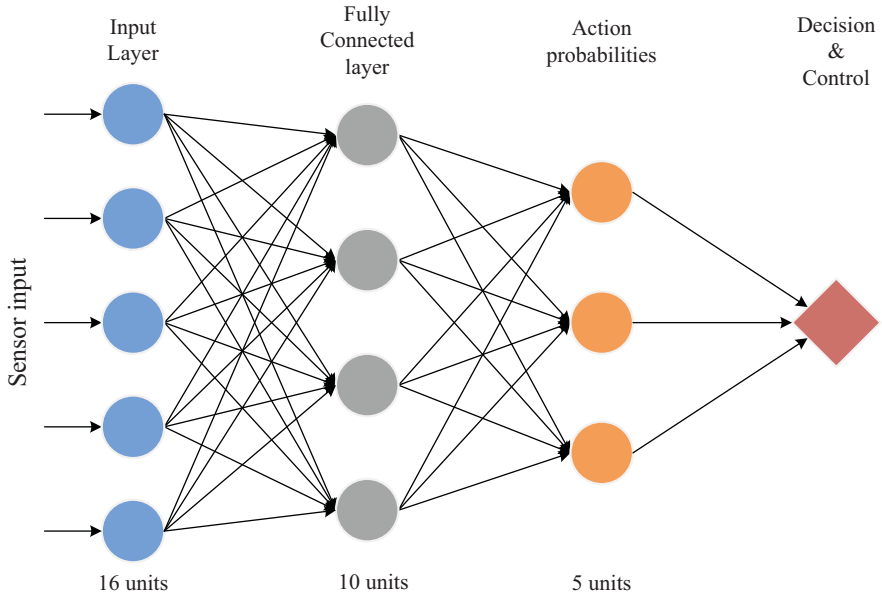



Figure 5.1: Nonlinear neural policy representation.

hidden layer. The weight $W^{(1)}$ is used to connect the input layer and the hidden layer, and similarly, the weight $W^{(2)}$ links the hidden layer and the output layer:

$$W^{(1)} = \begin{bmatrix} \omega_{1,0}^{(1)} & \dots & \omega_{1,k}^{(1)} \\ \vdots & \ddots & \vdots \\ \omega_{L_h,0}^{(1)} & \dots & \omega_{L_h,k}^{(1)} \end{bmatrix} \in \mathbb{R}^{L_h \times (k+1)}, \quad W^{(2)} = \begin{bmatrix} \omega_{1,0}^{(2)} & \dots & \omega_{1,L_h}^{(2)} \\ \vdots & \ddots & \vdots \\ \omega_{n,0}^{(2)} & \dots & \omega_{n,L_h}^{(2)} \end{bmatrix} \in \mathbb{R}^{n \times (L_h+1)}. \tag{5.8}$$

In our autonomous navigation problem, the network has 16 input units, 10 hidden units and 5 output units. $W^{(1)} \in \mathbb{R}^{10\times 17*}$, and similarly, $W^{(2)} \in \mathbb{R}^{5\times 11}$.

Keeping the weights $W^{(1)}$ and $W^{(2)}$ unchanged, according to Equations (3.19) - (3.21), we can easily calculate $\mathcal{G}(s_t)$ using FFNN with the parameter $\Omega = \{W^{(1)}, W^{(2)}\}$:

*The bias unit is included in the input layer, and thus the weight $w^{(1)}$ has 17 columns, but not 16. The same goes for the weight $w^{(2)}$.

$$
\begin{aligned}
z &= w^{(1)} \cdot x = w^{(1)} \cdot \begin{bmatrix} 1 \\ \phi_t \end{bmatrix}, \\
h &= \mathrm{sig}(z), \\
\mathcal{G}_\omega(s_t) &= \mathrm{sig}\left( w^{(2)} \cdot \begin{bmatrix} 1 \\ h \end{bmatrix} \right).
\end{aligned} \tag{5.9}
$$

where $\phi_t = \phi(s_t) \in \mathbb{R}^{k \times 1}$, and $\mathrm{sig}(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function, applied in both hidden and output units.

We also denote

$$\mathcal{G}_\omega(s_t, a_t) = [\mathcal{G}_\omega(s_t)]_i = Q(s_t, u_i), \quad \text{with } u_i = a_t.$$

### 5.4.3 Stochasticity of Policy

The stochastic policy is designed based on the neural representation, and is expressed by a (parametric) Boltzmann probability distribution $\pi_\Omega(s, a) = P(a|s; \Omega)$ that stochastically selects action $a$ in state $s$ according to NN weight $\Omega$:

$$\pi_\Omega(s_t, a_t) = P(a_t \,|\, s_t; \Omega) = \frac{\exp(\eta \mathcal{G}(s_t, a_t))}{\sum_{u \in \mathcal{A}} \exp(\eta \mathcal{G}(s_t))}. \tag{5.10}$$

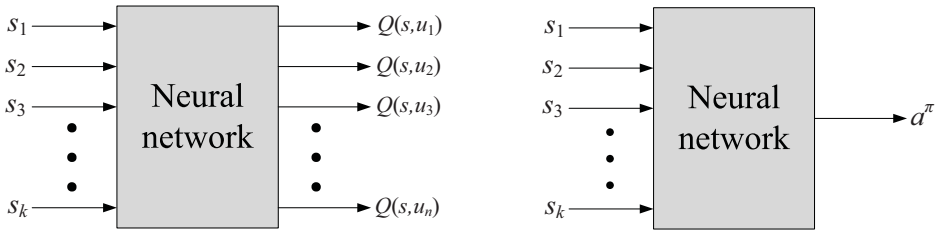
$\eta$ is the Boltzmann parameter that controls the stochasticity of action selection. If $\eta$ is low, all the choices have similar values, the policy outputs more stochastically. On the contrary, if $\eta$ is high, the action Q-values differ and the action with the highest Q-value is preferred to be picked. Thus, $P(a_t|s_t) \propto \exp(\eta Q(s_t, a_t)) > 0$.

Once the policy stops updating, the robot may start navigation by taking greedy action selection that selects actions according to the following equation is optimal:

$$a_t^\pi = \pi_\Omega(s_t, a_t) = \arg\max_{u \in A} \mathcal{G}(s_t). \tag{5.11}$$

Since the neural network can approximate any functions, our proposed method is greatly suitable for large state spaces. One may consider the neural policy representation as a black box which containing a nonlinear function between input features and output state-action values, see Figure 5.2(a).

Also, if we integrate the action selection strategy inside the black box, we may choose the selected action as the only output, see Figure 5.2(b).




(a) multiple outputs (b) single output

Figure 5.2: A black box structure of neural network.

## 5.5 Neural Inverse Reinforcement Learning

Inverse reinforcement learning algorithms always need to solve iteratively MDPs in order to optimize the reward function, and a model of environment is required in most IRL methods. Our method, NIRL, gets rid of this strict assumption and is a model-free method (Xia & El Kamel, 2015a,b). They also assume the reward function to be linear in the state features, and update all states to obtain a new policy in solving MDPs. With the adoption of our proposed nonlinear neural policy representation, NIRL only needs to update a subset of state-action space to achieve the generalization of unvisited regions, which largely reduces the learning time.

In our method, we do not assume the expert to be optimal. After obtaining the set of sub-optimal demonstration examples, we may pretreat the dataset via maximum a posteriori estimation (MAP) before proceeding them by NIRL.

### 5.5.1 Suboptimal Demonstration Refinement via Maximum a Posteriori Estimation

When an expert demonstrates one specific task, it is not guaranteed that he is always making decisions optimally. For instance, the expert may take different actions for a same state. This can be interpreted as the expert policy is assumed to be stochastic, even the actions may be stochastic. For example, we desire to steer the car to the left at 30°, however, due to the slippery road, the actual

steering angle turns out to be 60°. In these cases, we would like to pretreat the examples from the expert via maximum a posteriori estimation and finally refine them to be near-optimal.

Formally, given the expert examples $\mathcal{D}_E = \{(s_t, a_t)\}_{1 \le t \le M}$, there exist some state-action pairs that the same state correspond to multiple actions. The MAP is expressed by:

$$\hat{a}(s_t) = \arg\max_a P(s_t \,|\, a) P(a). \tag{5.12}$$

The likelihood $P(s_t \,|\, a)$ can be obtained by counting the frequency of $(s_t, a)$ in $\mathcal{D}_E$. The prior $P(a)$ can be of any predefined form. If we don't know the prior, we can assume $P(a)$ is uniformly distributed. In this case, we have $P(s_t \,|\, a)P(a) \propto P(s_t \,|\, a)$ , then $\hat{a}(s_t) = \arg\max_a P(s_t \,|\, a)$. A MAP estimation is reduced to a maximum likelihood (ML) estimation. Simply, for one particular state in all expert samples, the most likely action choice is the mode of all the actions corresponding to that state.

After the pretreatment of suboptimal examples, we can apply NIRL on them.

### 5.5.2 Model-free Maximum Margin Planning

NIRL is designed to learn a reward function from the expert demonstrations in the form of multiple trajectories. Assume that the reward $r_\theta(s, a) = \theta^\top f(s, a)$, and given a set of demonstrations $\mathcal{D}_E = \{\tau_E^{(i)}\}_{1 \le i \le N_T}$ with each trajectory $\tau_E^{(i)} = \{(s_t^{(i)}, a_t^{(i)})\}_{1 \le t \le L_i}$, we denote the empirical estimate for the expert feature expectations $\mu_E$ by

$$\hat{\mu}_E = \frac{1}{N_T} \sum_{i=1}^{N_T} \sum_{t=1}^{L_i} \gamma^{(t-1)} f(s_t^{(i)}, a_t^{(i)}). \tag{5.13}$$

The maximum margin planning method (MMP) (Ratliff *et al.*, 2006) is applied to find a policy $\pi$ that has feature expectations close to those of the expert, that is, $\|\mu^\pi - \mu_E\| < \epsilon$. It iteratively constructs a set of policies until a policy is found for which the distance between its value and the estimate of the expert's policy (using the expert's feature expectations) is smaller than a given threshold. If such a policy is not found, a new policy is generated as the optimal policy

corresponding to a reward function for which the expert does better by a margin than any of the policies previously found by the algorithm.

We use the max-margin algorithm in a model-free context, and we compare the values of the expert in every state in the recorded trajectories to the ones of learned policy in the same state. We can find the weight vector $\theta$ by solving the quadratic optimization problem:

$$J(\theta) = \max_{\theta} \sum_{i=1}^{N_T} \sum_{t=1}^{L_i} \left( Q_\theta^{\pi_E}(s_t^{(i)}, a_t^{(i)}) - \hat{Q}_\theta^{\pi}(s_t^{(i)}, a_t^{(i)}) \right)^2 + \lambda \|\theta\|_2, \tag{5.14}$$

where $\hat{Q}_\theta^{\pi}(s_t^{(i)}, a_t^{(i)})$ is an estimated $Q$-value of the learning agent in the current state $s_t^{(i)}$ of the expert. Estimating the value $\hat{Q}^{\pi}$ in a given state $s_t$ is calculated by using our neural policy representation.

The objective function $J(\theta)$ can be minimized by a subgradient descent method proposed in Ratliff *et al.* (2006, 2009b), i.e., $\theta \leftarrow \theta - \sigma \frac{\partial J(\theta)}{\partial \theta}$.

### 5.5.3 Neural Policy Iteration

After obtaining a reward function $r_\theta(s, a)$, we update the nonlinear neural policy iteratively through the backpropagation algorithm. That is realized by minimizing the neural network error $J(\Omega)$ via updating the weights $W^{(1)}$ and $W^{(2)}$. The network's error is the difference between its output for a given input and a target value, what the network is expected to output. The inputs are the state features and the outputs are the state-action values. The $Q$-values updated by RL algorithms are treated as the target values. $J(\Omega)$ is the cross-entropy cost function and defined as follows:

$$J(\Omega) = -\frac{1}{N} \sum_{t=1}^{N} \left[ \hat{Q}^{\pi} \circ \log Q^{\pi} + (1 - \hat{Q}^{\pi}) \circ \log (1 - Q^{\pi}) \right] + \frac{\lambda}{2N} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} \left( w_{j,i}^{(l)} \right)^2. \tag{5.15}$$

where

$$Q^{\pi} = Q^{\pi}(s_t, a_t) = \mathcal{G}(s_t, a_t).$$

$N$ is the total number of state-action pairs generated by the learned policy $\pi$. $L$ is the total number of layers in network, and $s_l$ is the number of units in layer $l$, excluding bias unit. The second term of $J(\Omega)$ is the regularization term aimed at avoiding overfitting or underfitting, and $\lambda$ is regularization parameter.

$\hat{Q}^{\pi}$ is calculated by the on-policy SARSA algorithm. The update rule is:

$$\hat{Q}^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) + \alpha[r_{\theta}(s_t, a_t) + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) - Q^{\pi}(s_t, a_t)]. \tag{5.16}$$

$\alpha$ is the learning rate, set between 0 and 1. Setting it to 0 means that the Q-values are never updated, hence nothing is learned. Setting a high value means that learning can occur quickly.

$\gamma$ is the discount factor with a range of 0 and 1. If $\gamma$ is close to 0, the robot will tend to consider immediate reward. On the contrary, if $\gamma$ approaches 1, the robot will take more future reward into account.

The gradient descent method is adopted to minimize the cost $J(\Omega)$ by

$$W^{(l)} \leftarrow W^{(l)} - \sigma \frac{\partial J(\Omega)}{\partial W^{(l)}}, \qquad \Omega = \{W^{(1)}, W^{(2)}, \cdots\}.$$

### 5.5.4 The Algorithm for Neural Inverse Reinforcement Learning

Given a set of demonstrations, if the expert is suboptimal, we should first decide if a MAP pretreatment is needed. Then after the refinement, we calculate the expert's feature expectations using Eq. (5.13).

The NIRL algorithm iteratively repeat three major steps:

1. Estimate the feature expectations of current learned policy,
2. Find current reward function using MMP,
3. Solve the MDP and calculate current optimal policy with respect to the reward function.

The complete NIRL algorithm for finding an optimal policy is presented in Algorithm 10.

**Algorithm 10** The NIRL algorithm

**Input:** the expert's feature expectations $\mu_E$.

Randomly generate an initial neural policy $\pi^{(1)}$, represented by its weights $\Omega^{(1)} = \{W^{(1)^{(1)}}, W^{(2)^{(1)}}\}$.
Set $i = 1$.
**while** $i > 0$ **do**
    Execute the current policy $\pi^{(i)}$ and generate a sequence of state-action pairs $\zeta^{(i)} = \{(s_t^{(i)}, a_t^{(i)})\}$, and compute $\mu^{(i)} = \mu(\Omega^{(i)})$.
    Minimize the objective function $t^{(i)} = \min_\theta J(\theta)$ in Eq. (5.14) s.t. $\|\theta\|_2 \leq 1$, and let $\theta^{(i)}$ be the value that attains this maximum.
    **if** $t^{(i)} \leq \epsilon$ **then**
        Terminate.
    **end if**
    Compute the rewards $r(s_t^{(i)}, a_t^{(i)}) = (\theta^{(i)})^\top f(s_t^{(i)}, a_t^{(i)})$, and using the SARSA algorithm, update the $Q$-values of the sequence $\zeta^{(i)}$.
    Apply the neural policy iteration to compute the current optimal neural weights $\Omega^{(i+1)} = \{W^{(1)^{(i+1)}}, W^{(2)^{(i+1)}}\}$, which represents the new policy $\pi^{(i+1)}$.
    Set $i = i + 1$.
**end while**

**Output:** a series of NN weights $\Omega^{(1)}, \Omega^{(2)}, \ldots$.

In the algorithm, the expert feature expectations $\mu_E$ are represented by its estimate value $\hat{\mu}_E$, i.e., $\mu_E = \hat{\mu}_E$.

The outputs of NIRL are also a series of policies:

$$\{\pi^{(1)}, \pi^{(2)}, \pi^{(3)}, \pi^{(4)}, \pi^{(5)}, \ldots\}.$$

We use these policies to predict the state-action pairs in the demonstrations $\mathcal{D}_E$, and we choose the policy that achieves the highest prediction accuracy as the optimal policy of NIRL.

Once learned, the NIRL can recover the reward for the current state space, and can predict the reward for any unseen state space within the domain of the features.

### 5.5.5 Expert Demonstrations

Two types of demonstrations are used in this paper. The first one is the human expert demonstrations. A human expert choose the best policy but may still not an optimal one. The second one is the computer expert demonstrations. A computer expert can always give optimal policies, and therefore a modified A* algorithm is applied to generate optimal policies. Another advantage of computer expert is that in some hazardous situations, human demonstrations cannot be available for the robot, so learning from computer experts bring less harm.

All the demonstration examples can be gathered either by the modified A* algorithm, or by the human expert, and NIRL algorithm learns the reward function. Finally a MDP solver finds a policy mapping from input of world states to output of robot actions.

#### 5.5.5.1 Computer-based Expert Demonstrations

The computer-based expert demonstrations are realized by the modified A* algorithm and we have presented it in Chapter 3.5.

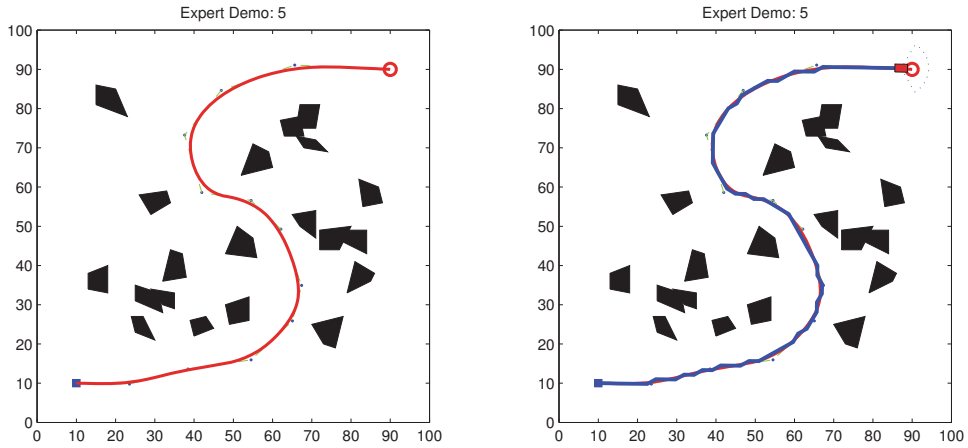
The expert's policy $\pi_E$ corresponds to the optimal deterministic policy.

#### 5.5.5.2 Human Demonstrations

A human expert can also demonstrate their navigating strategy by themselves. An expert can control the learning robot by teleoperation or directly show the robot how to navigate. We model the human demonstration in Figure 5.3. The human expert first gave some key points in the navigating map, and a smooth trajectory (Figure 5.3(a)) passing the key points is generated through interpolation techniques. The trajectory is regarded as the planned path for the robot. Then the robot follows the path to collect state-action pairs (Figure 5.3(b)). Obviously, a human expert may not always deliver an optimal policy.

## 5.6 Experimental results

The mobile robot (see Chapter 2.5) is represented by a rectangle-shaped robot. It is equipped with 9 sensors to observe the surrounding environment, as shown in Figure 2.6. The environment map has a size of 100 m $\times$ 100 m. The obstacles




(a) A trajectory given by a human expert

(b) Path following to collect data

Figure 5.3: A human expert demonstration.

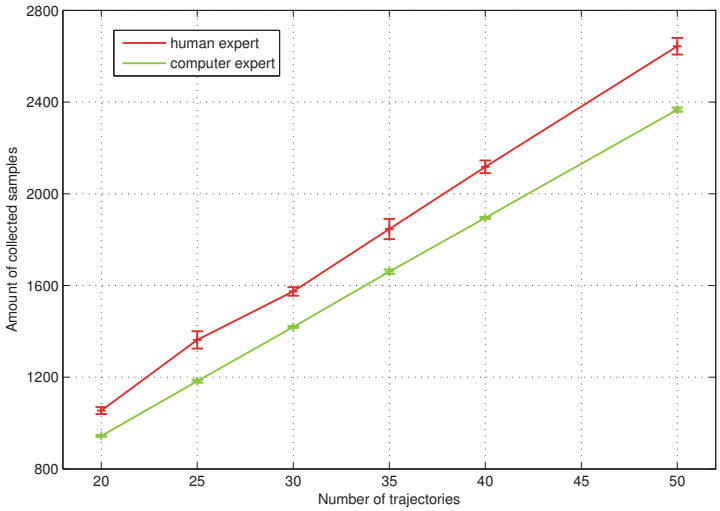
are randomly scattered in the environment and the robot has no prior knowledge of their numbers, sizes and positions. The initial position of the robot is placed at $(10, 10)$ and the target position, a red circle in the map, is found at $(90, 90)$. The velocity of the robot is fixed at 2 m/s. The mission of the robot is to start from the initial position and to find an optimal path to arrive at the target position without any collision with any obstacles. If no obstacles are detected, the robot is designed to move towards the target.

The neural network adopted in the experiments has three layers: 16 in the input layer, 8 in the hidden layer and 6 in the output layer. The inputs and outputs are presented above in Section 5.4.1. Some experiment parameters are selected as follows:

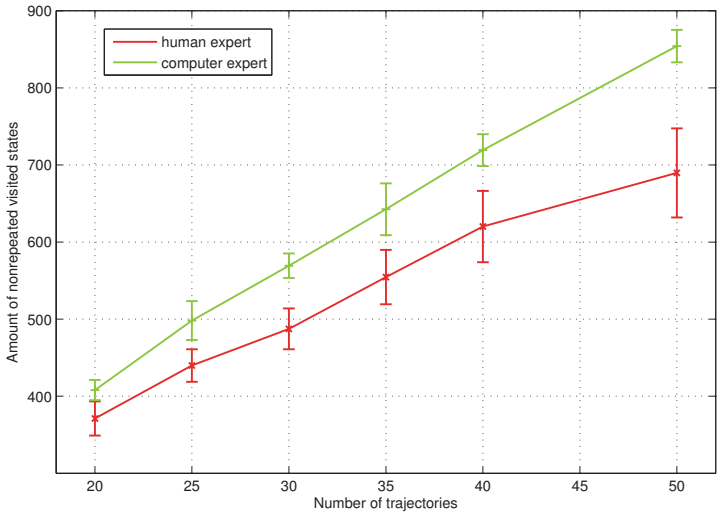
- Boundary distance: $d_{bou} = 10$ m,
- Warning distance: $d_{warn} = 5$ m,
- Learning rate of Q function: $\alpha = 0.8$,
- Learning rate of BPNN: $\sigma = 0.4$,
- Discount factor: $\gamma = 0.65$,

### 5.6.1 Comparison of Two Types of Demonstrations

In order to show the relationship between number of demonstrations and collected samples, different numbers of demonstrations were tested: 20, 25, 30, 35, 40 and 50. Recall that one sample is one state-action pair. Each number is conducted 10 times. The average results are shown in Figure 5.4(a). Averagely, one trajectory could contain 52-55 samples.




(a) The average amount of collected samples in different numbers of trajectories.



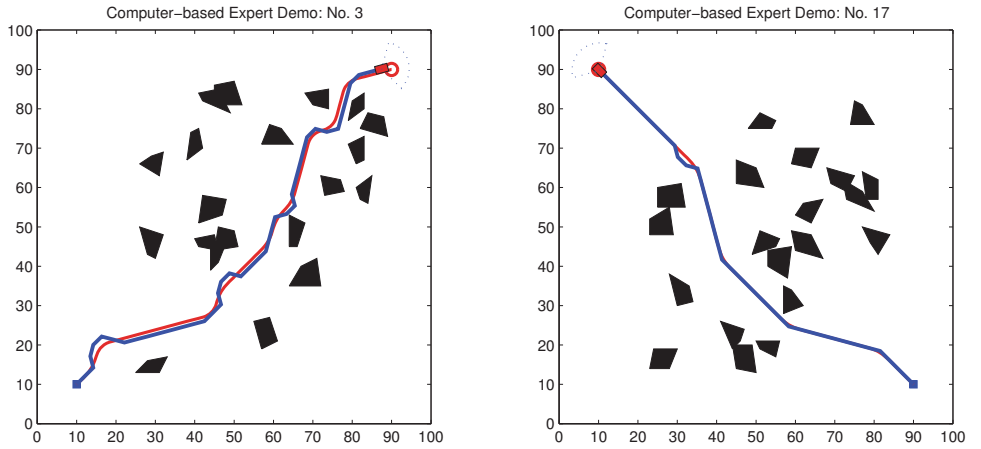

(b) The average amount of nonrepeated visited states in different numbers of trajectories.

Figure 5.4: Expert demonstrations in different numbers of trajectories.

The human's policies $\pi^E$ correspond to the sub-optimal stochastic policies. Due to the stochasticity, in the same states, there may exist many action possibilities. We concern how many different states have been visited during the demonstrations, therefore, we gave out the relation between the amount of non-repeated visited states and the number of demonstrations, as shown in Figure 5.4(b).

Averagely, one trajectory could contain 13-19 nonrepeated visited states.

When robot follows the expert trajectories, the actions that the robot take may be stochastic, that is, a suggested action may lead to a different action according a probability $P(\hat{a} \mid a)$, see Figure 5.5.




(a) An expert trajectory with several stochastic actions

(b) An expert trajectory with little stochastic actions

Figure 5.5: Expert demonstrations with stochastic actions.

In Figure 5.5(a), we can see a number of stochastic actions, while in Figure 5.5(b), there exists only one stochastic action.

### 5.6.2 Neural Inverse Reinforcement Learning

In this paper, 30 demonstrations are generated by modified A* algorithm. All of them have different configurations of random obstacle positions. The advantage of different maps is that the robot can always face new challenge and help gather more state-action combination. Some of the human expert demonstrations are presented in Figure 5.6.

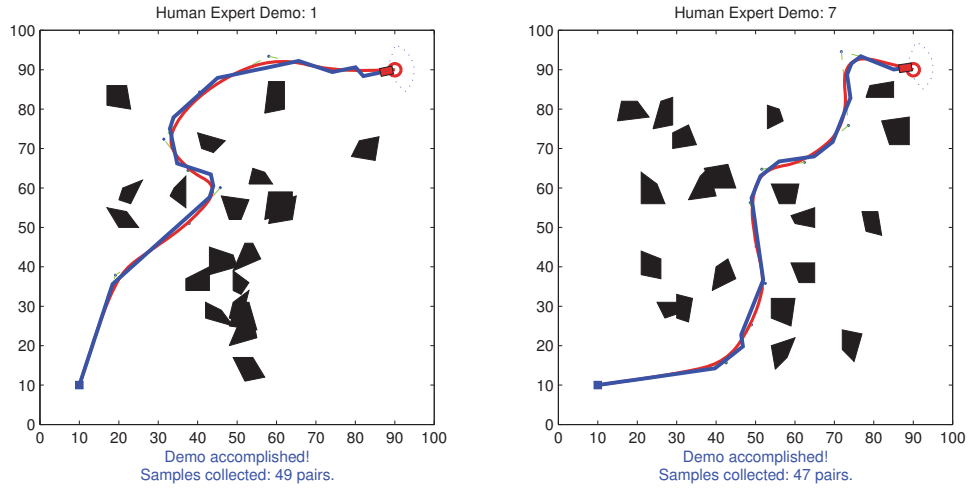



Figure 5.6: Human expert demonstrations.

It is observed that the computer-based demonstrations can always provide optimal paths, and this will greatly improve the robot learning performance compared to human teachers who cannot guarantee the quality of their demonstrations.

In each iteration of NIRL, an episode of 10 trajectories is generated using the current policy and all the sequences of state-action pairs are collected that are used for updating the policy. In each trajectory, the robot tries to navigate in different environments in order to assure the diversity of state-action pairs. A new episode will be started in the following three situations:

- The robot finds a collision free path to the target;
- The robot collides with an obstacle or the map borders;
- The robot runs out the moving steps.

The max-margin method is applied to learn the reward function by minimizing the margin function $J(\theta)$ in Equation 5.14.

Figure 5.7 showed the change of the margin in MMP during the learning episodes. We can tell the margin was stably diminishing towards 3. Hence, the margin between a learned policy was gradually approaching the expert policy, and this is exactly the learning objective.

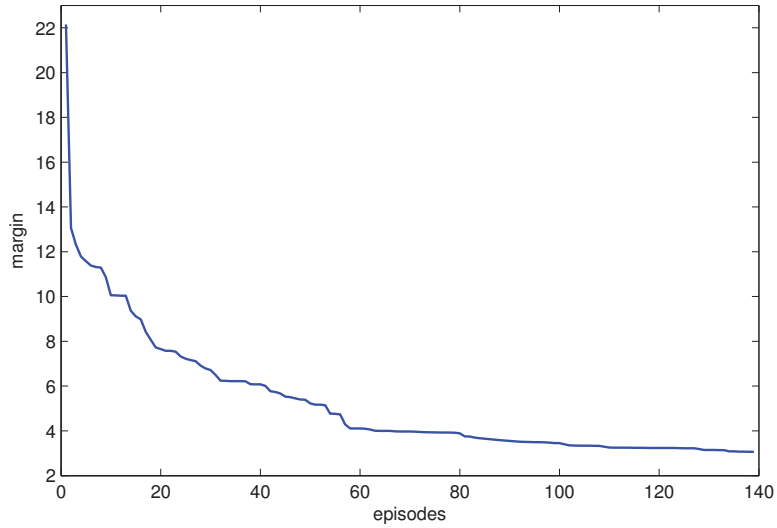



Figure 5.7: Changes of margin during the learning episodes.

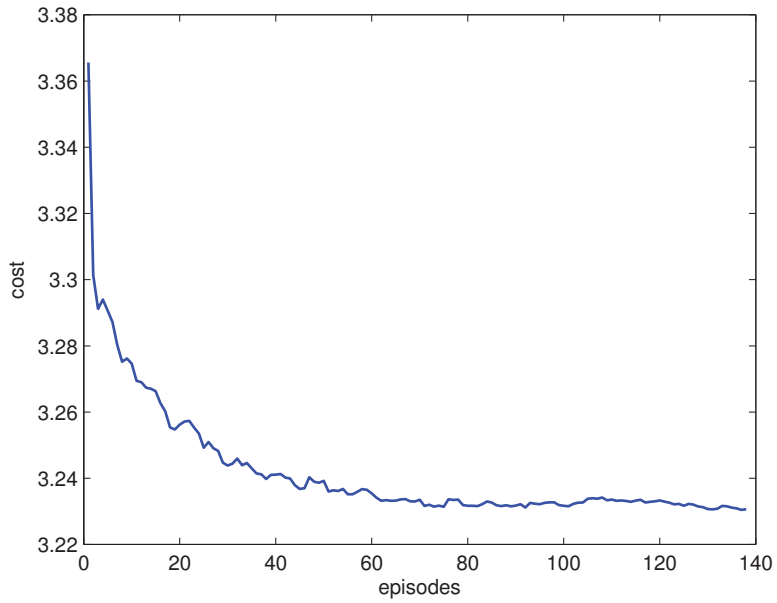



Figure 5.8: Changes of NN cost during the learning episodes.

The nonlinear neural policy is trained for the state-action pairs collected under the current policy. The goal is to minimize the cost $J(\Omega)$ in Equation (5.15) by iteratively updating the weights $W^{(1)}$ and $W^{(2)}$. The result is shown in Figure 5.8.

In the first 20 episodes, the cost decreases fast and then becomes smaller

stably. Finally after 140 episodes, the cost converges and stays around 3.23. Once the training terminates, $W^{(1)}$ and $W^{(2)}$ stop updating and the learning process from demonstration is accomplished. It is now the time to examine the robot learning achievement.

### 5.6.3 Robot navigation in new unknown environments

In the navigation process, the weights $W^{(1)}$ and $W^{(2)}$ have converged to their optimal values. The mobile robot has learned how to behave in front of obstacles in unexpected environments. Some of the navigation results are shown in Figure 5.9.

It is observed that the paths that the robot chose may not be the optimal ones due to a lack of complete knowledge of environment map, but they are still fully acceptable and perfect enough to meet our expectation, especially in such complicated and unstructured environments.

A successful navigation is one in which the robot succeeds in arriving at the target position. We tried 100 times of autonomous navigation by using various number of expert demonstrations. We compare the rate of success with both computer expert demonstrations (CE) and human expert demonstrations (HE).

| Number of demos | 25 | 30 | 40 | 50 |
|---|---|---|---|---|
| Rate of success of CE (%) | 90 | 91 | 89 | 87 |
| Rate of success of HE (%) | 82 | 85 | 84 | 84 |

Table 5.1: Rate of success in different numbers of demos

We observe that both types of demonstrations have proved to be reliable instructors to the robot, and the computer expert did slightly better. Also more demonstrations may not achieve a higher rate of success, because more demonstrations bring more stochastic in expressing the policies.

In order to test the robustness of the proposed algorithm, we gave the robot six completely new navigation missions using the same weights with three different robot velocities, namely 1 m/s, 2 m/s and 3 m/s. 50 runs were conducted for each velocities. The results are shown in Figure 5.10.

With the increase of obstacles presented in the surrounding environment, the robot becomes harder to navigate successfully. It is a normal situation that a

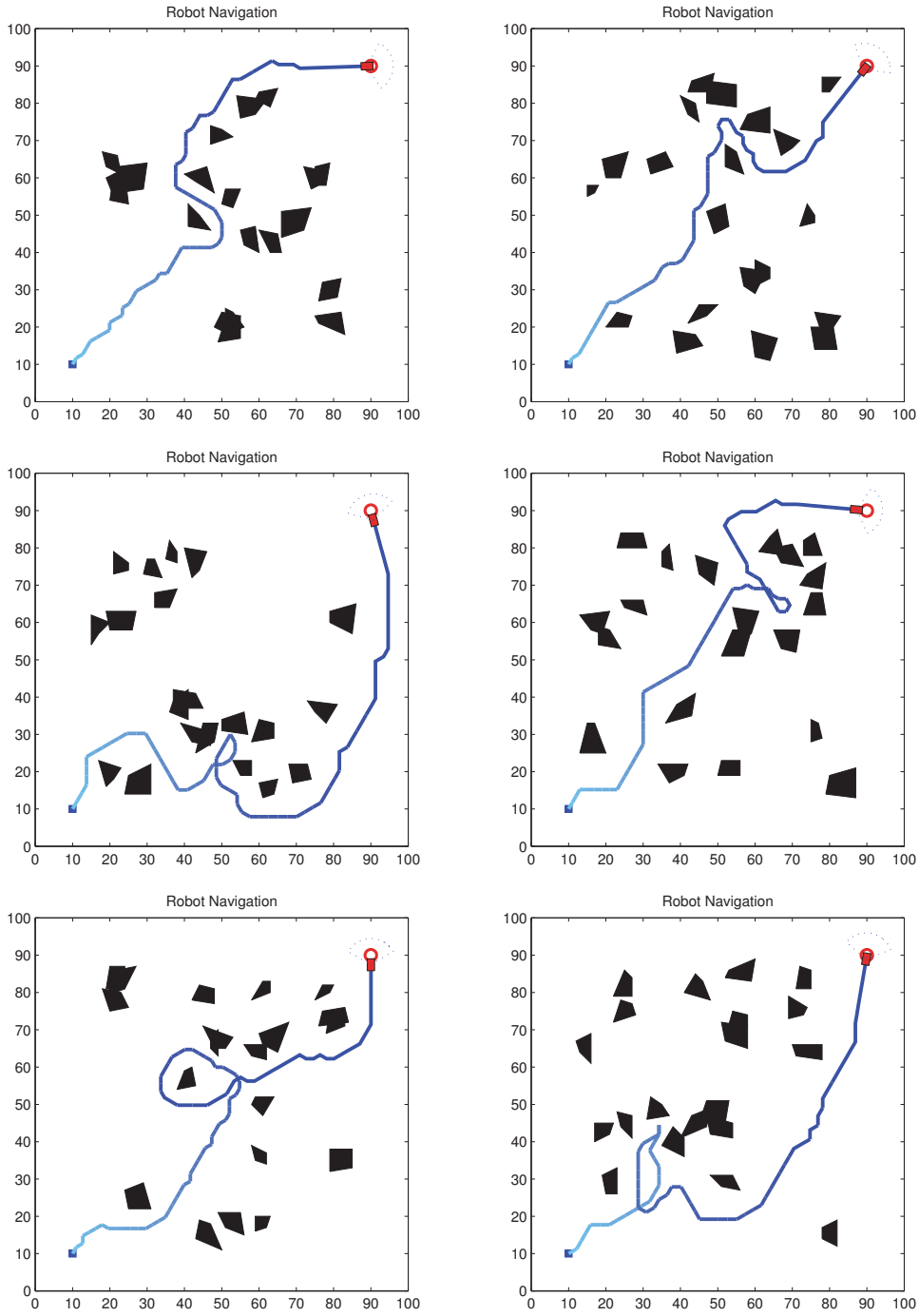



Figure 5.9: Robot navigation results in different environments.

bigger speed represents a need for quicker and more accurate reaction to the environment changes, especially in an completely unknown environment. Despite that, we can see that at least the robot could manage half of the navigation tasks

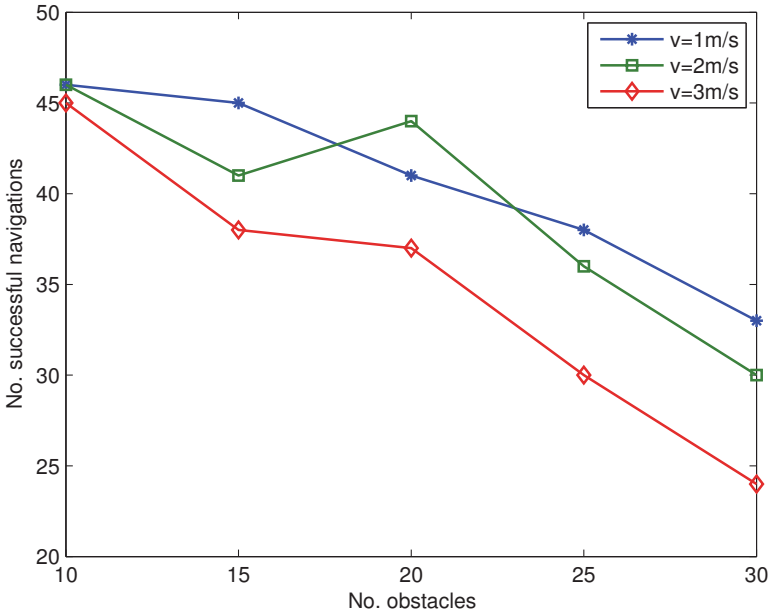



Figure 5.10: Comparison of numbers of successful navigations in the environments of different numbers of obstacles.

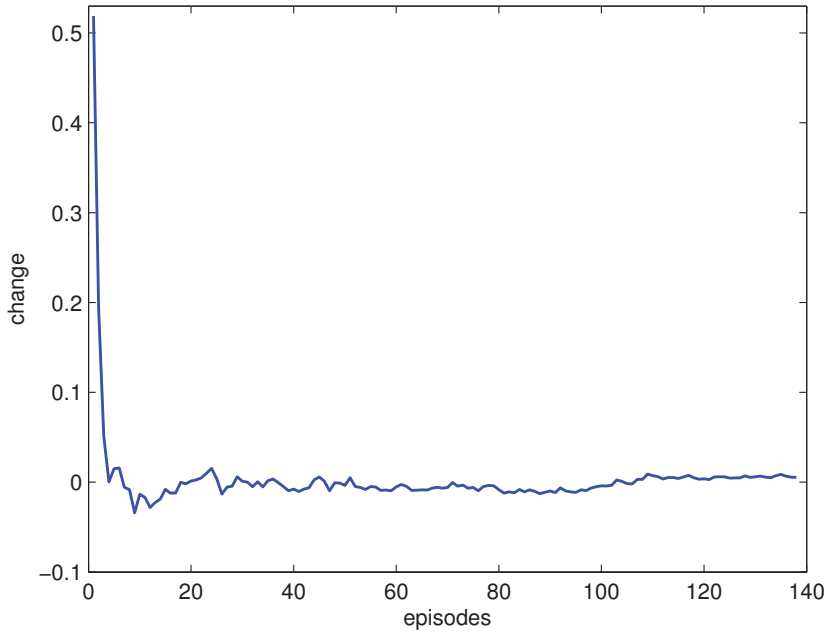
and in most cases, the robot did pretty good. Therefore, the above experiments have proven the feasibility and the stability of the proposed NIRL algorithm in different navigation environments and in different robot velocities.

### 5.6.4 Analysis of the weight changes

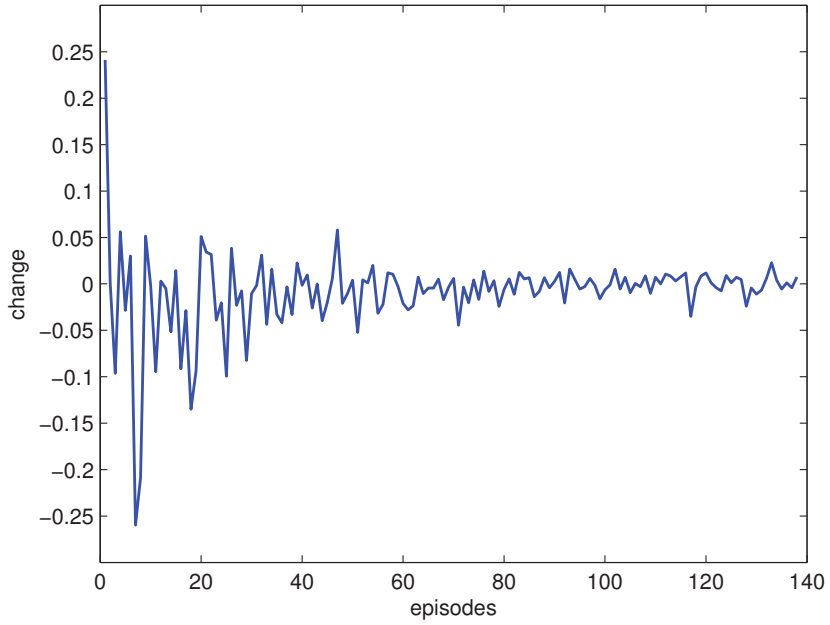
We analyzed the changes of neural policy weights $W^{(1)}$ and $W^{(2)}$ during the learning process using NIRL. The change in one episode is the difference between the weights before the learning and the ones after learning, i.e., $\|W^{(l)}_{\text{new}} - W^{(l)}\|_1$. The results are shown in Figure 5.11(a) and Figure 5.11(b). The X-axis represents the number of episode and the Y-axis is the weight change.

The changes of the weight $W^{(1)}$ fluctuated strong in the beginning episodes and then went quiet around 0 after 20 episodes. The overall trend of weight changes showed that the weight $W^{(1)}$ was turning stable as the learning process carried on and $W^{(1)}$ converged to its optimal value.

In the meantime, the changes of weight $W^{(2)}$ fluctuated more fiercely through all the learning process even though the changes went small after 60 episodes. That is because $W^{(2)}$ is directly related to the errors between the real outputs and the expected values in the neural network. Since the expected values are




(a) Changes of $W^{(1)}$ in learning process.




(b) Changes of $W^{(2)}$ in learning process.

Figure 5.11: Changes of neural policy weights during the learning process.

calculated in Equation (5.16), and in each learning episode, the robot needs to deal with a different environment, there are always errors between the real outputs and the expected values. Therefore, it is reasonable that the weight $W^{(2)}$ is more

difficult to converge to an optimal value. However, the overall trend still told us that $W^{(2)}$ was converging during the learning process, and the experiments showed that it did not affect the robot learning performance.

The changes of weights showed that NIRL converged to the optimal values after approximately 80 learning episodes, and the learned policy is close enough to the expert policy.

### 5.6.5 Autonomous Navigation in Dynamic Environments

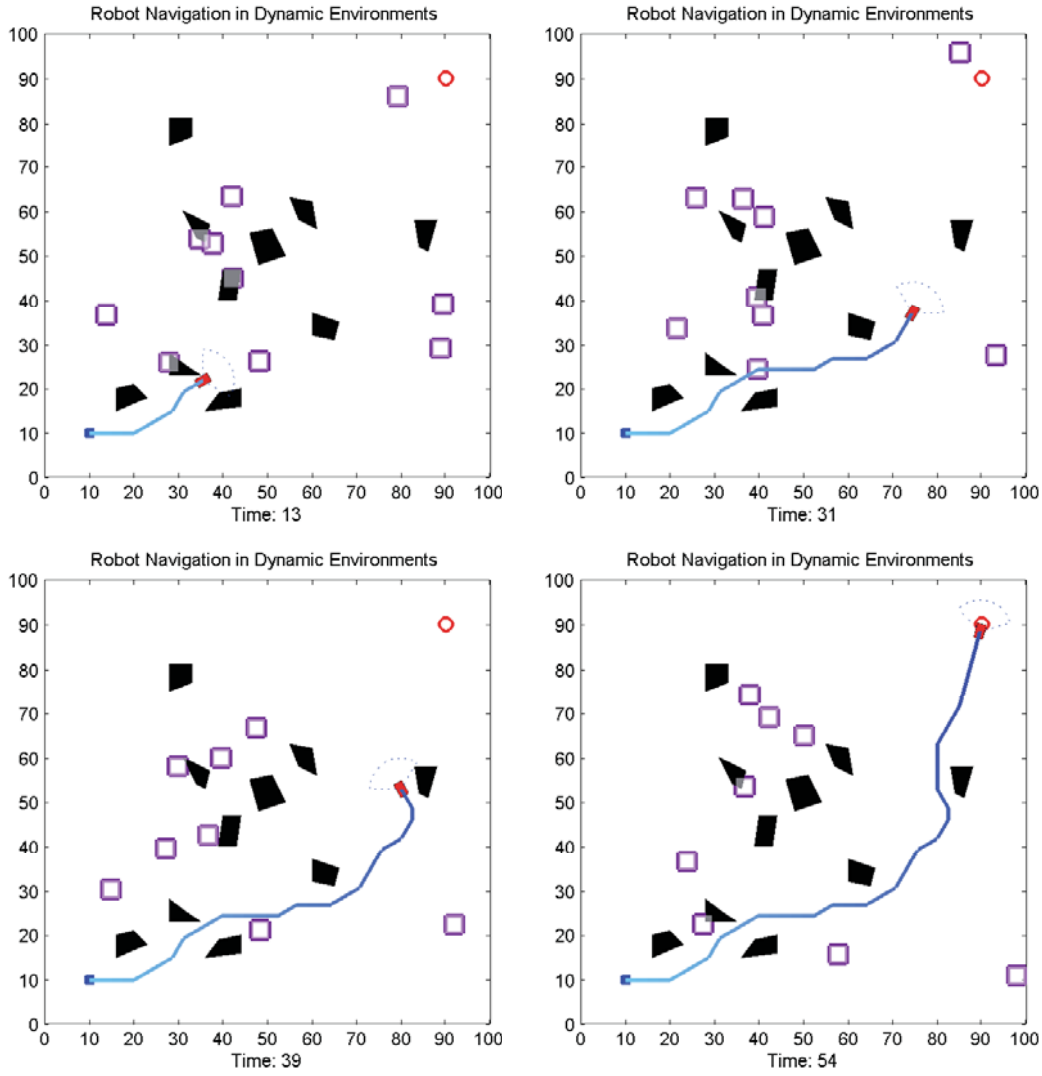



Figure 5.12: Autonomous robot navigation in a dynamic environment using NIRL.

We tested the NIRL algorithm in dynamic environments. The results are shown in Figure 5.12. In the environment, the black solid objects are static obstacles, and the purple rectangle ones are moving obstacles, whose moving directions are unpredictable.

There are 10 static obstacles and 10 moving obstacles. When the robot detected no obstacles around, it headed to the destination. From the experiment, we can see that at time 13, the robot met a static obstacle on its left and a moving obstacle in front of it, and the robot successfully avoided them and finally arrived at the destination.

Therefore, NIRL has proven its feasibility in the application of autonomous navigation in dynamic environments.

### 5.6.6 Discussions

We compare the NIRL algorithm with three major existing methods. The first one is the apprenticeship learning (AL) in (Abbeel & Ng, 2004; Abbeel *et al.*, 2008), the baseline of the family of inverse reinforcement learning. The second one is the relative entropy inverse reinforcement learning (RE), proposed in (Boularias *et al.*, 2011), a popular IRL algorithm using the theory of entropy. The last one is the structured classification inverse reinforcement learning (SCIRL), presented in (Klein *et al.*, 2012a), a recent work using max-margin method for a batch model-free learning algorithm.

We fixed the robot velocity at 2 m/s at the number of obstacles was 20. We conducted 25 runs for each learning method and each run was tested on 100 times of autonomous navigation.

For all the methods, we implemented the same computer expert. The comparison of the rate of success are shown in Table 5.2.

| | NIRL | AL | RE | SCIRL |
|---|---|---|---|---|
| Average rate (%) | 83 | 66 | 78 | 79 |
| Best rate (%) | 94 | 82 | 87 | 88 |
| Worst rate (%) | 69 | 43 | 61 | 60 |

Table 5.2: Comparison of the rate of success versus different IRL methods.

From the results, we can see that NIRL achieved the best rate of success. RE and SCIRL performed almost equally and slight worse than our NIRL algorithm. AL, as a pioneer algorithm, did the worst.

Therefore, the NIRL algorithm had a better performance in autonomous navigation tasks over the other three methods.

## 5.7 Conclusion

In the previous chapters, we have investigated two types of robot learning methods. The first one is to learn a policy from expert demonstrations, and the second one is to learn by itself by interacting with the surrounding unknown environments without any expert indication.

This chapter presents a third robot learning method, that the robot tries to understand a expert behaviors in executing one specific task by means of learning the underlying rewards from the expert demonstrations. This method is under the framework of inverse reinforcement learning. We developed our method, the neural inverse reinforcement learning algorithm. We first proposed a nonlinear neural policy representation by incorporating an artificial neural network. This explicit policy representation make the learning algorithm easy to implement. NIRL is also a method that does not assume the expert to be optimal. We proposed to applied maximum a posteriori to pretreat the suboptimal samples and later use the refined samples to NIRL. Then we adopted the maximum margin method to learn the reward through minimizing the maximum margin between the expert policy and the learned policy. Finally, we update the nonlinear neural policy using the learned reward function.

The experiment results show the feasibility and the robustness of the proposed NIRL algorithm, and he robot can complete autonomous navigation tasks safely in an unpredicted dynamic environment. The robot not only can imitate what the expert behave but also understand why the expert behave that way through learning the reward function. An autonomous mobile robot equipped with such learning ability is what we desire during the whole dissertation.

# Conclusions and Perspectives

In this chapter, we will conclude the dissertation by summarizing the main results and then present some perspectives of future research to complete and improve this work.

## Conclusions

Mobile robots are now playing a significant role in our society, the capability of autonomous navigation in dynamic environments is one of the fundamental skills. The requirement that the robot should be able to react to changes in environmental conditions led to the production of robust and reliable controllers that can deal with the uncertainty of the world. Since it is unlikely that we can foresee all potential real-world situations sufficiently accurate, a pre-programmed robot cannot meet the future requirements, especially when interaction with human is needed. The aim of this dissertation has been focused on empowering a mobile robots with an intelligent learning and adaptive ability that can cater to changing environments.

In this dissertation, we proposed three robot learning frameworks within different situations.

- Efficient policy learning from expert demonstrations. (Chapter 3)

  Human-beings are born with imitation ability, so should robots. When an expert demonstrates one specific task to robots, robots should know to imitate or learn how to behave or how to make decisions as their teacher does. Here comes our method of efficient policy learning from expert demonstrations. Different from previous work on learning from demonstration, we placed our focus on human experts are not available to robots. Instead, an improved version of A* is designed to stimulate expert examples. We

established a decision-making process and represented the policy by using neural network. This algorithm can be efficiently applied to autonomous navigation tasks. (Xia *et al.*, 2015)

- Self-learning in autonomous navigation using neural network based Q-learning. (Chapter 4)

  An intelligent robot should also learn to make decisions even without any demonstrations. That is realized by interacting with the environment of tasks. Based on these assumptions, we proposed neural network based Q-learning. Since we have proved that a neural network could be an appropriate policy representation, we improved the Q-learning algorithm by incorporating this representation. Results showed that our method greatly improved the learning performance and increase the convergence speed. (Xia & El Kamel, 2014a,b,c, 2015d)

- Learning reward from multiple examples via neural inverse reinforcement learning. (Chapter 5)

  Experts may not always demonstrate optimal policies. In these cases, inverse reinforcement learning provides a way to learn reward corresponding to each state. However, the generalization of undemonstrated states affects the performance of algorithm. We proposed the neural inverse reinforcement learning, which aligned our interest in neural network. Our method improved the performance on learning a reward function. This algorithm could be successfully applied to autonomous navigation, and as well as other applications. (Xia & El Kamel, 2015a,b)

All the above robot learning methods are model-free algorithms that are more suitable in real applications since a model of an unknown world is impossible to acquire.

## Future Work

First of all, this dissertation has provided theoretical foundations on mobile robot learning and achieved reliable results in simulators. Tests on real vehicles in outdoor environments could be conducted in order to carry the robot learning a step forward.

Second, an intelligent robot control system relies on the various sensors that capture the environmental information. With the development of robot sensors, an robot should be able to learn a control strategy from the rich variety of sensory data, especially those of visual and audio inputs. The ability of learning from a complexity of sensor data will not only endow the robot intelligence but also enhance the robot interaction with humans.

Third, robot learning should also be developed in multi-robot systems. More and more labor work would be done by a group of robots, and thus it would be necessary to pursue robot learning techniques to predict the behavior and better interact with other robots.

Last but not least, the deep reinforcement learning method has been presented by applying covolutional neural network (CNN) (Mnih *et al.*, 2013). As part of deep learning family, CNN has shown its power in image recognition and computer vision, therefore, researches in deep reinforcement learning will surely bring new breakouts to robotics.

# Résumé Étendu en Français

## Introduction

Les robots modernes sont conçus pour aider ou remplacer les humains à effectuer des opérations complexes, des planifications, ou des contrôles de tâches. Les exemples contiennent la manipulation d'objets, l'assistance d'experts dans une variété de professions, la navigation dans des environnements extérieurs, l'exploration des territoires inconnus, et la conduite dans les zones urbaines. Il est généralement un processus compliqué pour la conception d'un schéma de contrôle pour ces robots afin d'exécuter ces tâches, même pour les personnes spécialisées dans la programmation de robots. Elles doivent prendre en compte délibérément les spécifications de toutes les situations que le robot peut rencontrer et elle nécessitent à la création nouveau et différent à la main pour chaque tâche particulière. Ce genre de pré-programmation manuelle est généralement un travail intense et ennuyeux. Plutôt que de pré-programmer un robot pour toutes les tâches, il serait plus utile si le robot pourrait apprendre ces tâches par eux-mêmes.

## Movitation de la Recherche

La capacité de navigation autonome joue un rôle important dans la des voîtures autonomes sans conducteur de l'état de l'art. Remontée en 2003, La Defense Advanced Research Projects Agency (DARPA) du département de la Défense des États-Unis a organisé le "Grand Challenge" pour stimuler le développement des technologies nécessaires pour créer les véhicules terrestres sans pilote et autonomes capables de parcourir un cours substantielle hors route dans une période limitée. Le défi nécessite pour les véhicules de naviguer dans un circuit de 142

miles à travers du désert de Mojaves en ne pas surpassant 10 h. La première compétition a eu lieu le 13 mars, 2004. 15 véhicules ont y participé, mais malheureusement, aucune n'ont pu terminé plus de 5% de l'ensemble du cours. Par conséquent, le relancement d'un deuxième événement "DARPA Grand Challenge" était prévue le 8 octobre 2005. Cinq des 23 véhicules ont conquis le cours avec succès, et "Stanley" de Stanford (voir Figure 1) a été couronné le champion avec un résultat de 6 h 53 min. Cette voiture robotisée était une étape importante dans la quête des voitures autonomes modernes.


Figure 1: Stanley: le champion du DARPA Grand Challenge 2005. (Thrun *et al.*, 2006)

Deux ans plus tard, le "DARPA Urban Challenge" a eu lieu le 3 novembre, 2007, et a appelé à des véhicules autonomes de conduire 97 km à travers une maquette d'environnement urbaine en moins de 6 heures, en interagissant avec des obstacles et des autres véhicules en mouvement et aussi en respectant tous les règlements de la circulation. La véhicule "Boss" a été déclaré la gagnante (Urmson *et al.*, 2008) et "Junior" (Montemerlo *et al.*, 2008) a remporté la deuxième place. Ces vésicules ont également été considérées comme le premier prototype de la voiture sans conducteur de Google.

Traditionnellement, un programmeur humain utilisait son propre compréhension sur les tâches désirées et dressait un plan à l'avance pour coder un contrôleur de robot qui permettait de répondre à toutes les situations où le robot pouvait faire face. Si des erreurs ou de nouvelles circonstances apparaissaient, le robot devait être programmé à nouveau.

Motivé par les défis du DARPA et les voîtures sans conducteur de Google, cette thèse se concentre sur doter les robots la capacité d'apprendre de nouvelles techniques et d'améliorer leur compétences autonomes. Par ce moyen, les robots peuvent prendre de décisions intelligentes et adapter à des incertitudes et des changements imprévus dans les environnements stochastiques et dynamiques.

Cette thèse étudie le pilotage intelligent de robots dans les tâches de navigations autonomes et examiner l'apprentissage des robots dans trois aspects.

## Apprentissage efficace d'une politique par démonstrations d'experts

Dans le premier aspect, nous considérons l'apprentissage des robots par démonstrations d'experts. Cette méthode est inspirée par l'instinct humain de l'imitation. Les humains sont nées avec la capacité d'imitation, et les robots doivent donc aussi posséder cette capacité.

Nous fournissons un robot des exemples d'une tâche faite par un expert, le robot observe ces exemple et apprend une politique efficace à partir de ces données et de généraliser sur toutes les situations potentielles qui ne sont pas données dans les exemples. Dans la navigation autonome, une série de trajectoires sont données au robot, le robot apprend à construire une stratégie de contrôle afin d'accomplir la navigation autonome dans un environnent inconnu et dynamique (voir Figure 2).

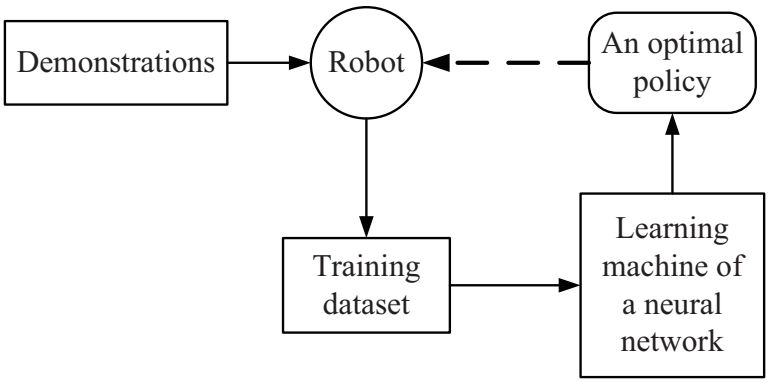



Figure 2: Un schéma simple de l'apprentissage d'une politique.

Nous avons intégré un mécanisme d'inférence en appliquant un réseau de neurones. Après avoir obtenu les démonstrations d'un expert, le robot sélectionne une base de données valables, et cette base est envoyée vers le réseau de neurones pour s'entraîner. L'algorithme de rétropropagation du gradient est appliqué pour minimiser itérativement une fonction d'entropie croisée. Notre méthode montre que avec un nombre acceptable de démonstrations, la structure d'apprentissage peut converger vers une politique optimale dans un nombre d'épisodes pas grand, et aussi le robot peut acquérir les compétences de navigation indépendants dans un nouveau environnent dynamique.

En plus, nous proposons un expert informatique basé sur une variante de l'algorithme A*. Cet expert peut offrir des trajectoires dans les cas extrêmes où les démonstrations humaines sont indisponibles.

L'avantage de cette méthode d'apprentissage efficace d'une politique est que le designer n'ont pas besoin de pré-programmer un robot précisément. L'utilisateurs peuvent facilement demander les robots d'exécuter un comportement par leur démontrer comment le faire.

## Apprentissage par renforcement sous des politiques stochastiques

Dans le deuxième aspect, nous considérons l'apprentissage de robots sans démonstrations d'experts et l'application dans la navigation autonome. C'est pour traiter les situations où les humains ont très peu d'expérience préalable, et donc aucunes démonstrations seraient être fourni. Un robot intelligent doit aussi posséder la compétence d'apprendre par soi-même à prendre des décisions sans aucune direction.

Nous proposons une méthode d'apprentissage en ligne via d'expérience accumulée qui est basée sur l'apprentissage par renforcement (voir Figure 3). Elle est réalisée par les interactions exploratoires entre le robot et son environnement. L'algorithme de Q-learning est une méthode qui ne nécessite aucun modèle initial du système, et Q-learning peut être appliqué pour trouver une suite d'actions associées à des états d'un Processus de décision markovien. Nous utilisons Q-learning pour mettre à jour les Q-valeurs dans chaque étape de temps. Différente que les autres méthodes concernant Q-learning qui utilisent les exemples venants

d'une seule interaction, notre méthode accumule les exemples venant de toutes les interactions dans une base de donnée, ainsi que leurs Q-valeurs renouvelées. La base est utilisée par un réseau de neurones afin de optimiser itérativement une politique de contrôle en temps réel. Pendant le processus d'interaction, le robot doit balancer l'exploration et l'exploitation, donc son politique est traitée stochastique.

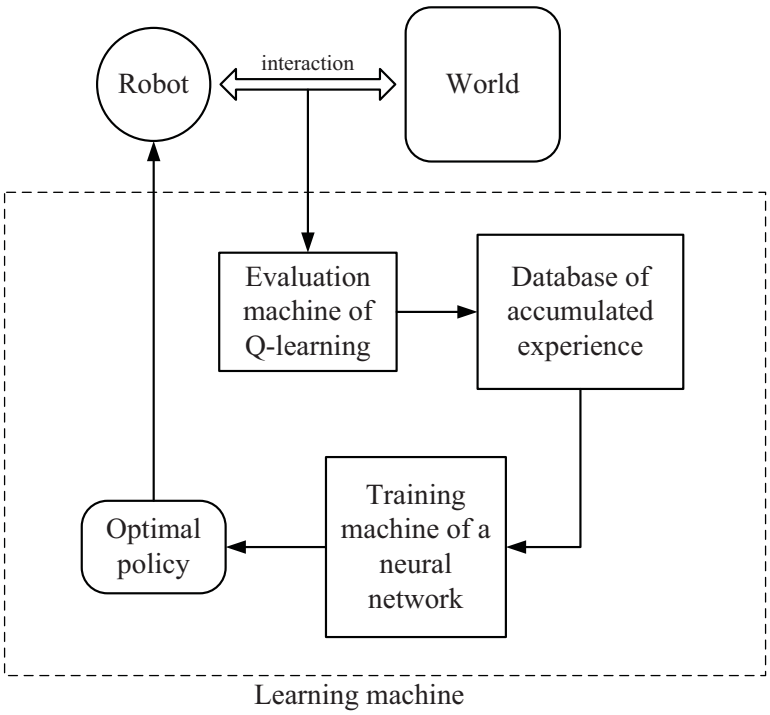



Figure 3: Un schéma simple de l'apprentissage en ligne via d'expérience accumulée.

Cette méthode évite l'inconvénient de celles dans la littérature qui abandonnent la plupart de données collectées pendant le processus d'interaction. Aussi, comme cette méthode améliore la politique en temps réel, sa performance est plus stable que celle de l'apprentissage en batch. Les expériences montrent le succès d'appliquer notre méthode à la navigation autonome et dépense moins de temps pour trouver une politique optimale.

# Apprentissage de la fonction de récompense via une représentation non-linéaire de la politique neuronale

Dans le troisième aspect, nous considérons que le robot apprend à comprendre les comportements d'un expert qui fait des démonstrations. Il est réalisé par apprendre les récompenses potentielles dans les états via l'apprentissage par renforcement inverse. C'est parce que une fonction de récompense n'est pas normalement facile à préciser et imiter directement un expert poserait le problème de l'imprécision si l'expert acte de manière non-optimale. Par conséquent, étant donné les démonstrations, les robots ne doivent pas seulement apprendre à associer les états et les actions, mais aussi essayer de comprendre les récompenses.

Nous proposons une méthode, l'apprentissage neuronale par renforcement inverse (voir Figure 4).

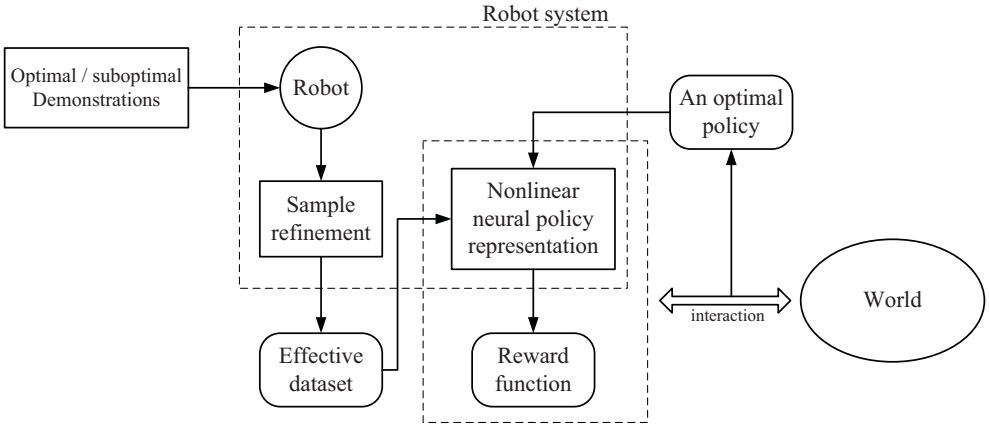



Figure 4: Un schéma simple de l'apprentissage de la fonction de récompense.

Une représentation non-linéaire de la politique neuronale est tout d'abord développée afin d'établir une connexion explicite entre les états et les actions, et un réseau de neurones est également intégré pour généraliser les actions sur les états non visités parmi les démonstrations. Ensuite la méthode proposée de l'apprentissage neuronale par renforcement inverse est appliquée à apprendre la fonction de récompense. Après avoir obtenu cette fonction, nous pouvons d'avantage optimiser itérativement la politique neuronale via les interactions progressives entre le robot et l'environnement et finalement déduire une politique optimale. Afin de traiter les démonstrations non-optimales, un raffinement est désigné à pré-procéder les démonstrations en profitant du maximum a posteriori.

Les résultats expérimentaux ont montré une performance fiable et la robustesse dans la tâche de navigation autonome en utilisant notre méthode proposée.

# Conclusions et Perspectives

Cette thèse contribue à la conception de commande intelligente afin de réaliser l'apprentissage des robots mobiles durant la navigation autonome. Les résultats expérimentaux ont confirmé l'efficacité et la robustesse de toutes les méthodes que nous avons proposé, et en plus, l'application de ces méthodes peut largement soulager du travail fastidieux de préprogrammer un robot manuellement.

## Contributions

L'objectif de cette thèse est centrée sur l'habilitation pour les robots mobiles des capacités d'apprentissage intelligent et d'adaptation qui permet de répondre à des environnements changeants. Les contributions sont développées dans trois domaines:

1. Dans les situations où les démonstrations optimales sont fournies, nous avons proposé la méthode d'apprentissage efficace d'une politique par démonstrations d'experts (Chapître 3). Cette méthode produit une performance stable et agréable avec un nombre acceptable de démonstrations. Elle économise aussi beaucoup de temps pendant le processus d'apprentissage.

2. Dans les situations où les humains manquent d'experience préalable, nous avons proposé la méthode d'apprentissage en ligne via d'expérience accumulée (Chapître 4). Cette méthode permet d'apprendre par soi-même une stratégie de contrôle et de traiter l'environnement progressivement en un certain nombre d'épisodes inférieur a la littérature. Notre méthode réussit une performance fiable et stable.

3. Dans les situations où les démonstrations fournies ne sont pas toujours optimales, nous avons proposé la méthode d'apprentissage neuronale par renforcement inverse (Chapître 5). Cette méthode peut efficacement apprendre la fonction de récompense en dessous de toute l'espace d'états. Cela permet

de mieux comprendre les comportements d'experts et ensuite calculer une politique de contrôle optimale, même si les exemples sont non-optimaux.

## Perspectives

Tout d'abord, cette thèse a fourni les théories fondementales sur l'apprentissage de robots mobiles et a obtenu des résultats fiables dans les simulateurs. Tests sur véhicules réelles dans des environnements extérieurs pourraient être menées afin de réaliser l'apprentissage de robots un pas en avant. En plus, d'autre applications pourraient aussi être étudié.

Deuxièmement, un système de contrôle du robot intelligent repose sur les différents capteurs qui perçoivent l'environnement. Avec le développement de capteurs, un robot doit être capable d'apprendre une stratégie de pilotage à partir de la variété riche de données sensorielles, en particulier celle des entrées audio et visuelles. La capacité d'apprendre à partir d'une complexité des données de capteurs non seulement dotera l'intelligence du robot, mais aussi améliorer l'interaction de robot avec les humains.

Troisièmement, l'apprentissage de robot devrait également être développé dans les systèmes de multi-robots. Puisque de plus en plus de travaux seraient effectués par un groupe de robots, il serait donc nécessaire de poursuivre les techniques d'apprentissage de robot pour prédire les comportements d'autre robots en coopération et mieux interagir avec eux.

Dernièrement, la méthode de l'apprentissage profond par renforcement a été présenté par l'application du réseau neuronal covolutional (CNN, *convolutional neural network* en anglais) (Mnih *et al.*, 2013). Dans le cadre de la famille de l'apprentissage profond, CNN a montré sa puissance en reconnaissance d'image et de vision par ordinateur, par conséquent, des recherches en l'apprentissage profond par renforcement apporteront certainement de nouveaux avancements à la robotique.

# Bibliography


Abbeel, P. & Ng, A.Y. (2004). Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, 1, ACM.

Abbeel, P. & Ng, A.Y. (2005). Exploration and apprenticeship learning in reinforcement learning. In *Proceedings of the 22nd international conference on Machine learning*, 1–8, ACM.

Abbeel, P., Coates, A., Quigley, M. & Ng, A.Y. (2007). An application of reinforcement learning to aerobatic helicopter flight. *Advances in neural information processing systems*, **19**, 1.

Abbeel, P., Dolgov, D., Ng, A.Y. & Thrun, S. (2008). Apprenticeship learning for motion planning with application to parking lot navigation. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, 1083–1090, IEEE.

Abbeel, P., Coates, A. & Ng, A.Y. (2010). Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*.

Alissandrakis, A., Nehaniv, C.L. & Dautenhahn, K. (2002). Imitation with alice: Learning to imitate corresponding actions across dissimilar embodiments. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, **32**, 482–496.

Andrieu, C., De Freitas, N., Doucet, A. & Jordan, M.I. (2003). An introduction to mcmc for machine learning. *Machine learning*, **50**, 5–43.

## BIBLIOGRAPHY


Argall, B.D., Chernova, S., Veloso, M. & Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and autonomous systems*, **57**, 469–483.

Atkeson, C.G. & Schaal, S. (1997). Robot learning from demonstration. In *Machine Learning (ICML), International Conference on*, vol. 97, 12–20.

Bagnell, J.A., Bradley, D., Silver, D., Sofman, B. & Stentz, A. (2010). Learning for autonomous navigation. *Robotics & Automation Magazine, IEEE*, **17**, 74–84.

Bentivegna, D.C., Atkeson, C.G. & Cheng, G. (2004a). Learning from observation and practice using primitives. In *AAAI 2004 Fall Symposium on Real-life Reinforcement Learning*, Citeseer.

Bentivegna, D.C., Atkeson, C.G., UDE, A. & Cheng, G. (2004b). Learning to act from observation and practice. *International Journal of Humanoid Robotics*, **1**, 585–611.

Bertsekas, D.P. (1996). *Dynamic programming and optimal control*, vol. 1. Athena Scientific Belmont, Massachusetts.

Bertsekas, D.P. (1998). *Network Optimization: continuous and discrete methods*, vol. 8. Athena Scientific.

Billard, A. & Grollman, D. (2013). Robot learning by demonstration. *Scholarpedia*, **8**, 3824.

Billard, A., Calinon, S., Dillmann, R. & Schaal, S. (2008). Robot programming by demonstration. In *Springer handbook of robotics*, 1371–1394, Springer.

Bohren, J., Foote, T., Keller, J., Kushleyev, A., Lee, D., Stewart, A., Vernaza, P., Derenick, J., Spletzer, J. & Satterfield, B. (2008). Little ben: the ben franklin racing team's entry in the 2007 darpa urban challenge. *Journal of Field Robotics*, **25**, 598–614.

Boularias, A. & Chaib-Draa, B. (2013). Apprenticeship learning with few examples. *Neurocomputing*, **104**, 83–96.

Boularias, A., Kober, J. & Peters, J.R. (2011). Relative entropy inverse reinforcement learning. In *International Conference on Artificial Intelligence and Statistics*, 182–189.

Boularias, A., Krömer, O. & Peters, J. (2012). Structured apprenticeship learning. In *Machine Learning and Knowledge Discovery in Databases*, 227–242, Springer.

Bradtke, S.J. & Barto, A.G. (1996). Linear least-squares algorithms for temporal difference learning. *Machine Learning*, **22**, 33–57.

Breazeal, C., Hoffman, G. & Lockerd, A. (2004). Teaching and working with robots as a collaboration. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 3*, 1030–1037, IEEE Computer Society.

Carreras, M., Yuh, J., Batlle, J. & Ridao, P. (2005). A behavior-based scheme using reinforcement learning for autonomous underwater vehicles. *Oceanic Engineering, IEEE Journal of*, **30**, 416–427.

Chernova, S. & Veloso, M. (2007). Confidence-based policy learning from demonstration using gaussian mixture models. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, 233, ACM.

Choi, J. & Kim, K.E. (2011a). Inverse reinforcement learning in partially observable environments. *The Journal of Machine Learning Research*, **12**, 691–730.

Choi, J. & Kim, K.E. (2011b). Map inference for bayesian inverse reinforcement learning. In *Advances in Neural Information Processing Systems*, 1989–1997.

Choi, J. & Kim, K.E. (2012). Nonparametric bayesian inverse reinforcement learning for multiple reward functions. In *Advances in Neural Information Processing Systems*, 305–313.

Coates, A., Abbeel, P. & Ng, A.Y. (2008). Learning for control from multiple demonstrations. In *Proceedings of the 25th international conference on Machine learning*, 144–151, ACM.

Cobo, L.C., Subramanian, K., Isbell, C.L., Lanterman, A.D. & Thomaz, A.L. (2014). Abstraction from demonstration for efficient reinforcement learning in high-dimensional domains. *Artificial Intelligence*, **216**, 103–128.

Cummins, M. & Newman, P. (2007). Probabilistic appearance based navigation and loop closing. In *Robotics and automation, 2007 IEEE international conference on*, 2042–2048, IEEE.

Dimitrakakis, C. & Rothkopf, C.A. (2012). Bayesian multitask inverse reinforcement learning. In *Recent Advances in Reinforcement Learning*, 273–284, Springer.

Duan, Y., Liu, Q. & Xu, X. (2007). Application of reinforcement learning in robot soccer. *Engineering Applications of Artificial Intelligence*, **20**, 936–950.

El-Fakdi, A. & Carreras, M. (2013). Two-step gradient-based reinforcement learning for underwater robotics behavior learning. *Robotics and Autonomous Systems*, **61**, 271–282.

Ernst, D., Geurts, P. & Wehenkel, L. (2005). Tree-based batch mode reinforcement learning. In *Journal of Machine Learning Research*, 503–556.

Fabiani, P., Fuertes, V., Piquereau, A., Mampey, R. & Teichteil-Königsbuch, F. (2007). Autonomous flight and navigation of vtol uavs: from autonomy demonstrations to out-of-sight flights. *Aerospace Science and Technology*, **11**, 183–193.

Fard, M.M. & Pineau, J. (2009). Mdps with non-deterministic policies. In *Advances in neural information processing systems*, 1065–1072.

Gosavi, A. (2009). Reinforcement learning: A tutorial survey and recent advances. *INFORMS Journal on Computing*, **21**, 178–192.

Harmon, M.E. & Harmon, S.S. (1996). Reinforcement learning: A tutorial. *WL/AAFC, WPAFB Ohio*, **45433**.

He, H., Eisner, J. & Daume, H. (2012). Imitation learning by coaching. In *Advances in Neural Information Processing Systems*, 3149–3157.

Helmick, D.M., Roumeliotis, S.I., Cheng, Y., Clouse, D.S., Bajracharya, M. & Matthies, L.H. (2006). Slip-compensated path following for planetary exploration rovers. *Advanced Robotics*, **20**, 1257–1280.

Hester, T., Quinlan, M. & Stone, P. (2012). Rtmba: A real-time model-based reinforcement learning architecture for robot control. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, 85–90, IEEE.

Huang, B.Q., Cao, G.Y. & Guo, M. (2005). Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance. In *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, vol. 1, 85–89, IEEE.

Jain, A., Wojcik, B., Joachims, T. & Saxena, A. (2013). Learning trajectory preferences for manipulators via iterative improvement. In *Advances in Neural Information Processing Systems*, 575–583.

Jaradat, M.A.K., Al-Rousan, M. & Quadan, L. (2011). Reinforcement based mobile robot navigation in dynamic environment. *Robotics and Computer-Integrated Manufacturing*, **27**, 135–149.

Kaelbling, L.P., Littman, M.L. & Moore, A.W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, 237–285.

Kaelbling, L.P., Littman, M.L. & Cassandra, A.R. (1998). Planning and acting in partially observable stochastic domains. *Artificial intelligence*, **101**, 99–134.

Klein, E., Geist, M. & Pietquin, O. (2012a). Batch, off-policy and model-free apprenticeship learning. In *Recent Advances in Reinforcement Learning*, 285–296, Springer.

Klein, E., Geist, M., Piot, B. & Pietquin, O. (2012b). Inverse reinforcement learning through structured classification. In *Advances in Neural Information Processing Systems*, 1007–1015.

Klein, É., Piot, B., Geist, M. & Pietquin, O. (2012c). Structured classification for inverse reinforcement learning. *Journal of Machine Learning Research*, **2012**, 1–14.

Knudson, M. & Tumer, K. (2011). Adaptive navigation for autonomous robots. *Robotics and Autonomous Systems*, **59**, 410–420.

Kober, J. & Peters, J. (2010). Imitation and reinforcement learning. *Robotics & Automation Magazine, IEEE*, **17**, 55–62.

Kober, J., Bagnell, J.A. & Peters, J. (2013). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 0278364913495721.

Kolter, J.Z. & Ng, A.Y. (2009). Policy search via the signed derivative. In *Robotics: science and systems*.

Kormushev, P., Calinon, S. & Caldwell, D.G. (2010). Robot motor skill coordination with em-based reinforcement learning. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, 3232–3237, IEEE.

Kunz, C., Murphy, C., Camilli, R., Singh, H., Bailey, J., Eustice, R., Jakuba, M., Nakamura, K.I., Roman, C., Sato, T. *et al.* (2008). Deep sea underwater robotic exploration in the ice-covered arctic ocean with auvs. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, 3654–3660, IEEE.

Lagoudakis, M.G. & Parr, R. (2003). Least-squares policy iteration. *The Journal of Machine Learning Research*, **4**, 1107–1149.

Levine, S., Popovic, Z. & Koltun, V. (2010). Feature construction for inverse reinforcement learning. In *Advances in Neural Information Processing Systems*, 1342–1350.

Levine, S., Popovic, Z. & Koltun, V. (2011). Nonlinear inverse reinforcement learning with gaussian processes. In *Advances in Neural Information Processing Systems*, 19–27.

Levine, S., Finn, C., Darrell, T. & Abbeel, P. (2015). End-to-end training of deep visuomotor policies. *arXiv preprint arXiv:1504.00702*.

Li, C., Zhang, J. & Li, Y. (2006). Application of artificial neural network based on q-learning for mobile robot path planning. In *Information Acquisition, 2006 IEEE International Conference on*, 978–982, IEEE.

Lockerd, A. & Breazeal, C. (2004). Tutelage and socially guided robot learning. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, vol. 4, 3475–3480, IEEE.

Lopes, M., Melo, F. & Montesano, L. (2009). Active learning for reward estimation in inverse reinforcement learning. In *Machine Learning and Knowledge Discovery in Databases*, 31–46, Springer.

Melo, F.S., Lopes, M. & Ferreira, R. (2010). Analysis of inverse reinforcement learning with perturbed demonstration. In *ECAI*, 349–354.

Michini, B. & How, J.P. (2012). Bayesian nonparametric inverse reinforcement learning. In *Machine Learning and Knowledge Discovery in Databases*, 148–163, Springer.

Michini, B., Cutler, M. & How, J.P. (2013). Scalable reward learning from demonstration. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, 303–308, IEEE.

Miljković, Z., Mitić, M., Lazarević, M. & Babić, B. (2013). Neural network reinforcement learning for visual control of robot manipulators. *Expert Systems with Applications*, **40**, 1721–1736.

Mitchell, T.M. (1997). Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, **45**.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Moldovan, T.M. & Abbeel, P. (2012). Safe exploration in markov decision processes. In *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, 1711–1718.

Montemerlo, M., Becker, J., Bhat, S., Dahlkamp, H., Dolgov, D., Ettinger, S., Haehnel, D., Hilden, T., Hoffmann, G., Huhnke, B. *et al.* (2008). Junior: The stanford entry in the urban challenge. *Journal of field Robotics*, **25**, 569–597.

Mori, T., Howard, M. & Vijayakumar, S. (2011). Model-free apprenticeship learning for transfer of human impedance behaviour. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, 239–246, IEEE.

Mülling, K., Kober, J., Kroemer, O. & Peters, J. (2013). Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research*, **32**, 263–279.

Navarro-Guerrero, N., Weber, C., Schroeter, P. & Wermter, S. (2012). Real-world reinforcement learning for autonomous humanoid robot docking. *Robotics and Autonomous Systems*, **60**, 1400–1407.

Neu, G. & Szepesvári, C. (2007). Apprenticeship learning using inverse reinforcement learning and gradient methods. In *Proc. UAI*.

Ng, A. (2011). Sparse autoencoder. *CS294A Lecture notes*, **72**.

Ng, A.Y. (2006). Reinforcement learning and apprenticeship learning for robotic control. In *Algorithmic Learning Theory*, 29–31, Springer.

Ng, A.Y. & Russell, S.J. (2000). Algorithms for inverse reinforcement learning. In *Machine Learning (ICML), International Conference on*, 663–670.

Ng, A.Y., Harada, D. & Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, vol. 99, 278–287.

Ng, A.Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E. & Liang, E. (2006). Autonomous inverted helicopter flight via reinforcement learning. In *Experimental Robotics IX*, 363–372, Springer.

Nicolescu, M.N. & Matarić, M.J. (2001). Learning and interacting in human-robot domains. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, **31**, 419–430.

Nicolescu, M.N. & Matarić, M.J. (2003). Natural methods for robot task learning: Instructive demonstrations, generalization and practice. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, 241–248, ACM.

O'kane, J.M., Tovar, B., Cheng, P. & LaValle, S.M. (2005). Algorithms for planning under uncertainty in prediction and sensing. *Autonomous Mobile Robots: Sensing, Control, Decision-Making, and Applications, Series in Control Engineering*, 501–547.

Peters, J., Mülling, K. & Altun, Y. (2010). Relative entropy policy search. In *AAAI*.

Piot, B., Geist, M. & Pietquin, O. (2013). Learning from demonstrations: Is it worth estimating a reward function? In *Machine Learning and Knowledge Discovery in Databases*, 17–32, Springer.

Piot, B., Geist, M. & Pietquin, O. (2014a). Boosted and reward-regularized classification for apprenticeship learning. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, 1249–1256, International Foundation for Autonomous Agents and Multiagent Systems.

Piot, B., Geist, M. & Pietquin, O. (2014b). Boosted bellman residual minimization handling expert demonstrations. In *Machine Learning and Knowledge Discovery in Databases*, 549–564, Springer.

Poole, D.L. & Mackworth, A.K. (2010). *Artificial Intelligence: foundations of computational agents*. Cambridge University Press.

Puterman, M.L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edn.

Qiao, J., Hou, Z. & Ruan, X. (2008). Application of reinforcement learning based on neural network to dynamic obstacle avoidance. In *Information and Automation, 2008. ICIA 2008. International Conference on*, 784–788, IEEE.

Qiao, Q. & Beling, P.A. (2011). Inverse reinforcement learning with gaussian process. In *American Control Conference (ACC), 2011*, 113–118, IEEE.

Rajasekaran, S. & Pai, G.V. (2011). *Neural networks, Fuzzy logic and Genetic algorithms*. PHI Learning Private Limited.

Ramachandran, D. & Amir, E. (2007). Bayesian inverse reinforcement learning. *IJCAI*, **51**, 2586–2591.

Ratliff, N., Ziebart, B., Peterson, K., Bagnell, J.A., Hebert, M., Dey, A.K. & Srinivasa, S. (2009a). Inverse optimal heuristic control for imitation learning. AISTATS.

Ratliff, N.D., Bagnell, J.A. & Zinkevich, M.A. (2006). Maximum margin planning. In *Proceedings of the 23rd international conference on Machine learning*, 729–736, ACM.

Ratliff, N.D., Silver, D. & Bagnell, J.A. (2009b). Learning to search: Functional gradient techniques for imitation learning. *Autonomous Robots*, **27**, 25–53.

Riedmiller, M. (2005). Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005*, 317–328, Springer.

Riedmiller, M., Gabel, T., Hafner, R. & Lange, S. (2009). Reinforcement learning for robot soccer. *Autonomous Robots*, **27**, 55–73.

Robins, B., Dautenhahn, K., Te Boekhorst, R. & Billard, A. (2004). Effects of repeated exposure to a humanoid robot on children with autism. In *Designing a more inclusive world*, 225–236, Springer.

Rojas, R. (1996). *Neural networks: a systematic introduction*. Springer.

Rothkopf, C.A. & Dimitrakakis, C. (2011). Preference elicitation and inverse reinforcement learning. In *Machine Learning and Knowledge Discovery in Databases*, 34–48, Springer.

Rummery, G.A. & Niranjan, M. (1994). On-line q-learning using connectionist systems.

Saunders, J., Nehaniv, C.L. & Dautenhahn, K. (2006). Teaching robots by moulding behavior and scaffolding the environment. In *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*, 118–125, ACM.

Schaal, S. & Atkeson, C.G. (2010). Learning control in robotics. *Robotics & Automation Magazine, IEEE*, **17**, 20–29.

Shon, A.P., Verma, D. & Rao, R.P. (2007). Active imitation learning. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, vol. 22, 756, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Siegwart, R., Nourbakhsh, I.R. & Scaramuzza, D. (2011). *Introduction to autonomous mobile robots*. MIT press.

Silver, D., Bagnell, J. & Stentz, A. (2008). High performance outdoor navigation from overhead data using imitation learning. *Robotics: Science and Systems IV, Zurich, Switzerland*.

Silver, D., Bagnell, J.A. & Stentz, A. (2010). Learning from demonstration for autonomous navigation in complex unstructured terrain. *The International Journal of Robotics Research*.

Silver, D., Bagnell, J.A. & Stentz, A. (2012). Active learning from demonstration for robust autonomous navigation. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, 200–207, IEEE.

Singh, S., Jaakkola, T., Littman, M.L. & Szepesvári, C. (2000). Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, **38**, 287–308.

Smallwood, R.D. & Sondik, E.J. (1973). The optimal control of partially observable markov processes over a finite horizon. *Operations Research*, **21**, 1071–1088.

Sutton, R.S. & Barto, A.G. (1998). *Reinforcement learning: An introduction*. MIT press.

Sweeney, J.D. & Grupen, R. (2007). A model of shared grasp affordances from demonstration. In *Humanoid Robots, 2007 7th IEEE-RAS International Conference on*, 27–35, IEEE.

Syed, U. & Schapire, R.E. (2007). A game-theoretic approach to apprenticeship learning. In *Advances in neural information processing systems*, 1449–1456.

Syed, U. & Schapire, R.E. (2010). A reduction from apprenticeship learning to classification. In *Advances in Neural Information Processing Systems*, 2253–2261.

Syed, U., Bowling, M. & Schapire, R.E. (2008). Apprenticeship learning using linear programming. In *Proceedings of the 25th international conference on Machine learning*, 1032–1039, ACM.

Szepesvári, C. (2010). Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, **4**, 1–103.

Thomaz, A.L. & Breazeal, C. (2008). Teachable robots: Understanding human teaching behavior to build more effective robot learners. *Artificial Intelligence*, **172**, 716–737.

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G. *et al.* (2006). Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, **23**, 661–692.

Tian, D., Xia, C. & El Kamel, A. (2015). Behavior coordination and command fusion of autonomous mobile robot using fuzzy control method. In *2nd Conference on Embedded Systems, Computational Intelligence and Telematics in Control (CESCIT 2015)*, Paper 10, Maribor, Slovenia.

Tossou, A.C. & Dimitrakakis, C. (2013). Probabilistic inverse reinforcement learning in unknown environments. *arXiv preprint arXiv:1307.3785*.

Tzafestas, S.G. (2013). *Introduction to mobile robot control*. Elsevier.

Ude, A., Atkeson, C.G. & Riley, M. (2004). Programming full-body movements for humanoid robots by observation. *Robotics and autonomous systems*, **47**, 93–108.

Urmson, C., Anhalt, J., Bagnell, D., Baker, C., Bittner, R., Clark, M., Dolan, J., Duggins, D., Galatali, T., Geyer, C. *et al.* (2008). Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, **25**, 425–466.

van Lent, M. & Laird, J.E. (2001). Learning procedural knowledge through observation. In *Proceedings of the 1st international conference on Knowledge capture*, 179–186, ACM.

Wang, Y.H., Li, T.H.S. & Lin, C.J. (2013). Backward q-learning: The combination of sarsa algorithm and q-learning. *Engineering Applications of Artificial Intelligence*, **26**, 2184–2193.

Watkins, C.J. & Dayan, P. (1992). Q-learning. *Machine learning*, **8**, 279–292.

Watkins, C.J.C.H. (1989). *Learning from delayed rewards*. Ph.D. thesis, University of Cambridge England.

Wiering, M. & van Otterlo, M. (2012). *Reinforcement Learning: State-of-the-art*, vol. 12. Springer Science & Business Media.

Xia, C. & El Kamel, A. (2014a). An intelligent method of mobile robot learning in unknown environments. In *International Conference on Computer Science and Information Technology (ICCSIT)*, C048, Barcelona, Spain.

Xia, C. & El Kamel, A. (2014b). Mobile robot navigation using neural network based q-learning. In *2014 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, paper 197, Bali, Indonesia.

Xia, C. & El Kamel, A. (2014c). Mobile robot navigation using neural network based q-learning. In *3rd International Conference on Control, Robotics and Informatics (ICCRI)*, M0014, Hong Kong.

Xia, C. & El Kamel, A. (2015a). Learning reward functions with non-linear neural policy representations. *Engineering Applications of Artificial Intelligence*, under review.

Xia, C. & El Kamel, A. (2015b). Neural inverse reinforcement learning in autonomous navigation. *Robotics and Autonomous Systems*, under review.

Xia, C. & El Kamel, A. (2015c). Online reinforcement learning from accumulated experience based on a nonlinear neural policy. *Expert Systems with Applications*, submitted.

Xia, C. & El Kamel, A. (2015d). A reinforcement learning method of obstacle avoidance for industrial mobile vehicles in unknown environments using neural network. In *2014 International Conference on Industrial Engineering and Engineering Management (IEEM)*, 671–675.

Xia, C., Tian, D. & El Kamel, A. (2015). Improving autonomous navigation in unknown environments via learning from demonstration. In *2nd Conference on Embedded Systems, Computational Intelligence and Telematics in Control (CESCIT 2015)*, Paper 63, Maribor, Slovenia.

Xu, X., Zuo, L. & Huang, Z. (2014). Reinforcement learning algorithms with function approximation: Recent advances and applications. *Information Sciences*, **261**, 1–31.

Yang, G.S., Chen, E.K. & An, C.W. (2004). Mobile robot navigation using neural q-learning. In *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, vol. 1, 48–52, IEEE.

Zhifei, S. & Joo, E.M. (2012). A review of inverse reinforcement learning theory and recent advances. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, 1–8, IEEE.

Ziebart, B.D., Maas, A.L., Bagnell, J.A. & Dey, A.K. (2008). Maximum entropy inverse reinforcement learning. In *AAAI*, 1433–1438.

Ziebart, B.D., Bagnell, J.A. & Dey, A.K. (2010). Modeling interaction via the principle of maximum causal entropy.

## Apprentissage Intelligent des Robots Mobiles dans la Navigation Autonome

**Résumé:** Les robots modernes sont appelés à effectuer des opérations ou tâches complexes et la capacité de navigation autonome dans un environnement dynamique est un besoin essentiel pour les robots mobiles. Dans l'objectif de soulager de la fastidieuse tâche de préprogrammer un robot manuellement, cette thèse contribue à la conception de commande intelligente afin de réaliser l'apprentissage des robots mobiles durant la navigation autonome. D'abord, nous considérons l'apprentissage des robots par démonstrations d'experts. Nous proposons d'utiliser un réseau de neurones pour apprendre hors-ligne une politique de commande à partir de données utiles extraites d'expertises. Ensuite, nous nous intéressons à l'apprentissage sans démonstrations d'experts. Nous utilisons l'apprentissage par renforcement afin que le robot puisse optimiser une stratégie de commande pendant le processus d'interaction avec l'environnement inconnu. Un réseau de neurones est également incorporé et une généralisation rapide permet à l'apprentissage de converger en un certain nombre d'épisodes inférieur à la littérature. Enfin, nous étudions l'apprentissage par fonction de récompenses potentielles compte rendu des démonstrations d'experts optimaux ou non-optimaux. Nous proposons un algorithme basé sur l'apprentissage par renforcement inverse. Une représentation non-linéaire de la politique est désignée et la méthode du max-margin est appliquée permettant d'affiner les récompenses et de générer la politique de commande. Les trois méthodes proposées sont évaluées sur des robots mobiles afin de leurs permettre d'acquérir les compétences de navigation autonome dans des environnements dynamiques et inconnus.

**Mots-clés:** Apprentissage automatique, Robots mobiles, Apprentissage par renforcement, Réseau de neurones, Navigation autonome, Apprentissage par démonstrations, Processus de décision markovien.

## Intelligent Mobile Robot Learning in Autonomous Navigation

**Abstract:** Modern robots are designed for assisting or replacing human beings to perform complicated planning and control operations, and the capability of autonomous navigation in a dynamic environment is an essential requirement for mobile robots. In order to alleviate the tedious task of manually programming a robot, this dissertation contributes to the design of intelligent robot control to endow mobile robots with a learning ability in autonomous navigation tasks. First, we consider the robot learning from expert demonstrations. A neural network framework is proposed as the inference mechanism to learn a policy offline from the dataset extracted from experts. Then we are interested in the robot self-learning ability without expert demonstrations. We apply reinforcement learning techniques to acquire and optimize a control strategy during the interaction process between the learning robot and the unknown environment. A neural network is also incorporated to allow a fast generalization, and it helps the learning to converge in a number of episodes that is greatly smaller than the traditional methods. Finally, we study the robot learning of the potential rewards underneath the states from optimal or suboptimal expert demonstrations. We propose an algorithm based on inverse reinforcement learning. A nonlinear policy representation is designed and the max-margin method is applied to refine the rewards and generate an optimal control policy. The three proposed methods have been successfully implemented on the autonomous navigation tasks for mobile robots in unknown and dynamic environments.

**Keywords:** Machine learning, Mobile robots, Reinforcement learning, Neural network, Autonomous navigation, Learning from demonstration, Markov decision processes.